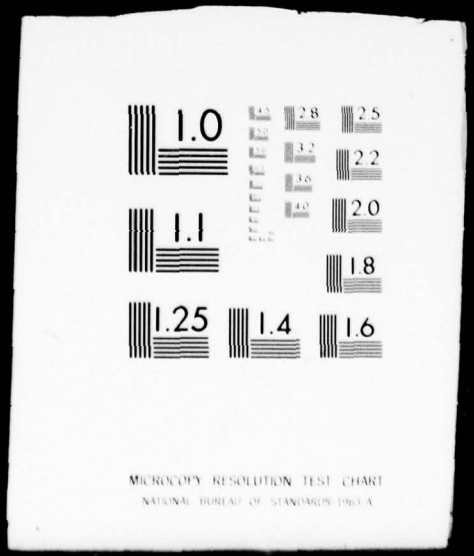
AD-A070 955 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE REPRESENTATION OF FAMILIES OF SOFTWARE SYSTEMS.(U)
APR 79 L W COOPRIDER F44620-73-C-0074
UNCLASSIFIED CMU-CS-79-116 AFOSR-TR-79-0732 NL

1 OF 3
AD A070955



CMU-CS-79-116

The Representation of Families of Software Systems

Lee W. Cooprider

14 April 1979

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) Unclassified

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AFOSR-TR-79-0732	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): THE REPRESENTATION OF FAMILIES OF SOFTWARE SYSTEMS.	5. TYPE OF REPORT & PERIOD COVERED: Interim rept.
6. PERFORMING ORG. REPORT NUMBER: CMU-CS-79-116	
7. AUTHOR(s): Lee W. Coopridder	8. CONTRACT OR GRANT NUMBER(s): F44620-73-C-0074, NSF-DCR74-24457
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61101E, A02466/7
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Blvd, Arlington, VA 22209	12. REPORT DATE: April 1979
	13. NUMBER OF PAGES: 194
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office): Air Force Office of Scientific Research (NM), Bolling AFB, D.C. 20332	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software system, family of systems, module interconnection, software engineering database, system description language, software construction, version, automated programming, maintenance.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Programmming languages are notations for the representation of algorithmic infromation they are tools for "programming-in-the-small" DeRe76 . System description languages are notations for "programmin-in-the-large". Because software systems often exist in several versions simultaneously, a system description language must accomodate parallel versions of systems and permit the natural expression of the information sharing among those versions.

The construction of software systems involves sequences of construction

continued

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403 081

20. Abstract (continued)
processes such as text editing, compilation, document production, linkage editing, and cross-reference generation. Automation of these processes has been inpeded by the use of inadequate models of software construction and maintenance. As a resoult, the enforcement of design decisions described in a system description language has been left to human agencies.

We develop a notation for describing the subsystem interconnections of entire systems, the differences between versions of those systems, and the mechanisms by which the systems are constructed. Subsystems are objects which provide a set of resources to other subsystems and require a set of resources that are supplied by other subsystems. Each interconnection network can be instantiated in several versions. Versions are organized hierarchically so that similar versions share part of their descriptions.

Detailed system construction processes, such as text editing, compilation and document gereration, are expressed in a functional form. Resources and source files are combined according to construction rules to create the concrete objects that are the tangible (executable or readable) form of a software system. The construction processes are controlled by the interconnection structure and version specifications in which they are defined.

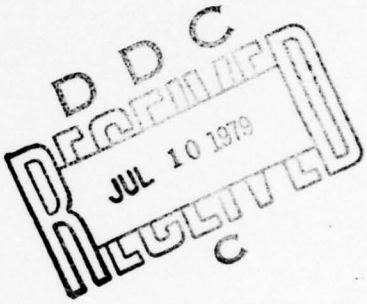
This representation is the basis for the design of a software construction database. The database manager automatically performs system construction processes, propagates modifications to system components, and maintains construction histories. The database user can establish invariants in the database by attaching policies to each database object; the policies supply a set of actions to be preformed when events in the database affect the object (e.g. a component has been modified).

An extended example is presented to demonstrate the applicability of this presentation to a real system. Several types of system construction problems are discussed, and directions for improvements to the notation outlined.

CMU-CS-79-1.

The Representation of Families of Software Systems

Lee W. Cooprider



Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213
ARPAnet address: Cooprider@CMU-10A

14 April 1979

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was funded in part by the National Science Foundation under Contract No. DCR74-244573, in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract No. F44620-73-C-0074, and in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, N.J.

Abstract

Programming languages are notations for the representation of algorithmic information; they are tools for "programming-in-the-small"[DeRe76]. System description languages are notations for "programming-in-the-large". Because software systems often exist in several versions simultaneously, a system description language must accomodate parallel versions of systems and permit the natural expression of the information sharing among those versions.

The construction of software systems involves sequences of construction processes such as text editing, compilation, document production, linkage editing, and cross-reference generation. Automation of these processes has been impeded by the use of inadequate models of software construction and maintenance. As a result, the enforcement of design decisions described in a system description language has been left to human agencies.

We develop a notation for describing the subsystem interconnections of entire systems, the differences between versions of those systems, and the mechanisms by which the systems are constructed. Subsystems are objects which provide a set of resources to other subsystems and require a set of resources that are supplied by other subsystems. Each interconnection network can be instantiated in several versions. Versions are organized hierarchically so that similar versions share part of their descriptions.

Detailed system construction processes, such as text editing, compilation and document generation, are expressed in a functional form. Resources and source files are combined according to construction rules to create the concrete objects that are the tangible (executable or readable) form of a software system. The construction processes are controlled by the interconnection structure and version specifications in which they are defined.

This representation is the basis for the design of a software construction database. The database manager automatically performs system construction processes, propagates modifications to system components, and maintains construction histories. The database user can establish invariants in the database by attaching policies to each database object; the policies supply a set of actions to be performed when events in the database affect the object (e.g. a component has been modified).

An extended example is presented to demonstrate the applicability of this representation to a real system. Several types of system construction problems are discussed, and directions for improvements to the notation outlined.

Keywords: software system, family of systems, module interconnection, software engineering database, system description language, software construction, version, automated programming, maintenance.

Acknowledgements

This dissertation was developed with the guidance of Nico Habermann and a thesis committee consisting of Mary Shaw, Charles Eastman and Anita Jones. Dave Parnas contributed to my general research direction and Frank DeRemer contributed a semester of valuable discussions early in this project. Dave Notkin and Mark Faust assisted with the implementations. The author benefitted from the personal and scientific excellence that characterizes the Carnegie-Mellon University Computer Science Department. The preparation of this document was facilitated by the Scribe program, a product of Brian Reid's experience and diligence. Various personal demons were banished with the assistance of Mame Novotny and other members of the Pittsburgh Re-evaluation Counseling Community.

Table of Contents

Table of Figures

Thesis Summary

1 Introduction

Recent software engineering research has addressed the problem of developing software that is correct, efficient, reliable, and modifiable. Software system implementors now have the guidance of programming methodologies and soon will be able to exploit the expressiveness of new programming languages. A central theme of this research is the decomposition and encapsulation of information.

The components of a software system that has been decomposed must be recombined to form operational systems. Many components participate in several systems, and may exist in several forms themselves. Research on subsystem interconnection languages has begun to illuminate the problem of representing the way in which decomposed systems are structured.

Many software systems exist in several versions simultaneously. Current subsystem interconnection notations do not adequately represent the close relationship between members of such system families. Without such a representation, the problem of maintaining parallel versions of systems is difficult if not intractable.

We develop an integrated representation for families of software systems that includes the subsystem interconnection information and permits the specification of multiple versions of systems and the control of shared information. We encompass the details of system component construction such that an automated software construction system can be built around the representation.

2 Development of Notation

The representation of software systems is divided into three levels. The first level describes subsystem interconnection in the style of previous interconnection languages. At the second level, each subsystem description can be instantiated; each instantiation corresponds to the intuitive notion of "version". Each version consists of several concrete objects such as source files, compiled programs, or documents. The construction of these objects is described in the third level of the notation and incorporates the traditional processes performed by compilers, editors and linkers.

1 Subsystem Interconnection

A subsystem is an object which provides a set of resources to other subsystems and requires a set of resources from other subsystems. A resource is some information, such as an algorithm, a data representation, a table, a grammar, a finite state machine, or an integer. The resources required by a subsystem can be supplied by constituent subsystems, which are textually nested in the definition of the original subsystem, or by external subsystems such as library facilities. In some cases, resources can also be supplied by an enclosing subsystem.

Here is a very simple example of a subsystem (PLOT) that provides a plotting facility using the trigonometric functions provided by another subsystem (TRIG):

subsystem *PLOT* **provides** *PlotRoutines* **requires** *TrigFcns* **external** *TRIG*
realization . . . **end** *PLOT*

A subsystem that is composed of a group of subsystems textually encloses that group. It

may supply a set of resources to the entire group by defining a resource environment. Resources provided by internal subsystems may be passed through to users of the enclosing subsystem. The following subsystem exemplifies these features. The subsystem provides two sets of operations on processes; each of those sets is actually provided by an internal subsystem. The enclosing subsystem obtains the definition of some operating system primitives from the OPS subsystem and provides them to both internal subsystems via a resource environment.

```
subsystem PROCESS provides DynamicProcesses, ProcessControls
    requires DynamicProcesses, ProcessControls
    environment OperatingSystemPrimitives external OPS
    subsystem DP provides DynamicProcesses . . . end DP
    subsystem PC provides ProcessControls . . . end PC
    realization . . . end PROCESS
```

2 System Instantiation

Each version of a subsystem adheres to the subsystem interconnection structure. However, there may be substantial variation in the contents and organization of each version of that subsystem. Individual versions may vary in implementation language, algorithmic strategy, capacity, specifications, measurability and so forth. There are three aspects to a version definition: the concrete objects which are used to comprise the version, the selection of versions for resource providing subsystems, and additional resource requirements specific to the version.

Each subsystem contains a tree of versions. Versions at the leaves of the tree associate resources with concrete objects. For example, the resource *TrigFcns* might be associated with a file containing Fortran subroutines for sine, cosine and arctangent functions. Intermediate versions isolate information shared by subsidiary versions. The entire tree is a powerful mechanism for controlling the relationship between the abstract resource domain and the concrete construction domain. Here is the TRIG subsystem from the previous example with a possible collection of versions. Let us assume that these programs are used in source form in Fortran, Algol and Pascal programs and that each version is stored in a separate file.

```
subsystem TRIG provides TrigFcns
    realization
        version Fortran
            version Source resources file(<Fortran Source>) end Source
            version Debug resources file(<Fortran Debug Source>) end Debug
            end Fortran
        version Algol
            version Standard resources file(<Standard Algol Source>) end Standard
            version Local resources file(<Local Algol Source>) end Local
            end Algol
        version Pascal resources file(<Pascal Source>) end Pascal
    end TRIG
```

Each subsystem uses resources from another collection of subsystems; it follows that each version of a subsystem uses resources from specific versions of those other subsystems. Each level in the version hierarchy selects a group of versions sufficient to completely specify the origin of resources used in the creation of concrete objects specified at that same level. The following subsystems describe users of the PLOT subsystem and include

version selections.

```
subsystem ARCHAIC provides Histogram requires PlotRoutines external PLOT
    realization
        version Rusty select PLOT=Backup.Fortran . . . end Rusty
    end ARCHAIC

subsystem MODERN provides SketchPad requires PlotRoutines external PLOT
    realization
        version Shiny select PLOT=Current.Pascal . . . end Shiny
    end MODERN
```

3 Construction

The concrete objects listed in the versions of subsystems must be constructed using the types of software tools normally available, such as editors, compilers and linkers. Each of those objects is defined in the notation at a point that corresponds to its use, such that objects used only in the construction of one version of the system are defined within that version. Each concrete object is the product of a "processor". Special processors *file* and *acquire* permit direct access to files and resources. A compiler or other general processor takes two kinds of parameters, concrete objects such as source files and output from other processors, and strings, such as compiler control parameters.

```
file(<Old Fortran Source>)
acquire(TrigFcns)
Fortran(file(<Old Fortran Source>))
Algol(file(<Old Algol Source>)) with "BoundsCheckOn"
```

Any place a concrete object is needed, the construction rule can be written. It is also possible to give names to concrete objects and then use the name as a parameter to a construction rule. Simple block structured scope on object names achieves the desired locality on most concrete objects.

Here is the same example modified to include the construction information. Suppose that the development version of these routines incorporate programs that translate Fortran to Algol and Pascal (including resolution of resources), and that <Plot Routine Source> is a file containing the Fortran program. FortranResolve is a program that inserts resources into Fortran program text, keying on a special comment indicator "C#".

```
subsystem PLOT provides PlotRoutines requires TrigFcns external TRIG
    realization
        version Backup
            version Fortran select TRIG=Fortran.Source
                resources FortranResolve(file(<Old Fortran Source>))
                end Fortran
            end Backup
        version Current
            version Fortran  select TRIG=Fortran.Source
                resources FortranResolve(file(<Current Fortran Source>))
                end Fortran
            end Current
```

```
        version Development
            concrete object Original=file(<Plot Routine Source>)
            version Fortran select TRIC=Fortran.Source
                resources FortranResolve(Original) end Fortran
            version Algol select TRIC=Algol.Source
                resources FortranToAlgol(Original) end Algol
            version Pascal select TRIC=Pascal.Source
                resources FortranToPascal(Original) end Pascal
            end Development
    end PLOT
```

4 Acquire Mechanism

The *acquire* mechanism converts a resource name into a string that contains the representation of the resource. In the file <Plot Routine Source> shown below, the phrase "Acquire(TrigFcns)" directs a Fortran processor to convert the resource name "TrigFcns" into a string containing program text by looking up the resource in the TRIG subsystem. It is possible to invoke the acquire function in the construction description as well as from within a processor (no example shown here).

```
C       Plot Routine Source File -- Fortran with Resource Requirements
C
        Subroutine Plot1(A,B,C)
        . . .
        End
C
        Subroutine Plot2(D,E,F)
        . . .
        End
C
C#      Acquire(TrigFcns)
C
C       End of Plot Routine Source File
```

3 Software Construction Database

The concepts described above are used in the design of a software construction database. The database consists of typed entries corresponding to subsystems, versions, concrete objects, and other linguistic structures. The database manager is capable of automating significant aspects of the software construction and maintenance process.

The user modifies the database objects interactively and invokes construction processes on versions of subsystems. The interposition of the database manager prevents uncontrolled manipulation of files, compilation steps and system descriptions. Although the user may request only that a particular component of a particular system be built, the database manager may determine that other objects must be built to satisfy the user request. Of special interest is the automatic construction of the representation of resources upon request from a processor.

The database manager is also able to automatically perform other maintenance tasks. Using instructions supplied by the user in the form of "policies", the database manager can propagate modifications to affected modules, check consistency of test results for modified programs, and notify the user of potential incompatibility in the system.

Two central mechanisms of the database are the *construct* facility, which controls the construction of a concrete object, and the *acquire* facility, which obtains the representation of a resource. The construction of a complex system component begins with the invocation of a processor. At some point, that processor may issue a call to *acquire* to obtain a resource definition. The database manager will construct the requested resource definition, using the *construct* function to invoke another processor; that construction may in turn request resource definitions. This stack-oriented construction model contrasts strongly with the conventional sequential model. The implementations of *acquire* and *construct* are presented in detail. Versions of both functions were implemented and ran on a simple database.

4 Example

Without an extended example, it is difficult to determine whether the notation is appropriate for a significant proportion of the system structures actually encountered by software engineers. A collection of examples are drawn from a real system, the software support for a scanline printer. Included in the collection are the top-level system description, a number of intermediate subsystems which provide substantial functionality, and a number of low-level resources which are widely used throughout the system.

The top-level subsystem collects together the resources provided by its constituent subsystems. It contributes no additional resources but serves to delimit the system intuitively perceived by the user community. Some resource provided by constituents are hidden from general usage and exploited only within the confines of the top-level system description.

One intermediate subsystem extracts pages from a document formatted for the scanline printer. The representation for this subsystem is augmented by the complete construction process for the source program. Two of the resources used by this program are encoded as macros, and one of those sets of macros is generated automatically from a tabular definition of part of the document format definition. Some of the abstract operations used by the original source program do not fit an abstract data type model.

One of the low level subsystems maintains a list of character sets used to describe the output of the scanline printer. This relatively simple subsystem enforces the consistency of the various representations of the character set list by generating each of the representations from a tabular representation. Policies attached to this table can cause the complete reconstruction of all programs that use the table any time a modification occurs.

The printer driver subsystem exemplifies several complex version relationships. One such problem arises because the driver can either operate the printer directly or queue requests for later processing. The command languages for the two versions of the driver are different, and therefore the data tables for the driver help command are different. The union of the command language keywords is stored in a file from which the relevant subsets are drawn during construction.

5 Conclusions

An integrated design/construction representation is feasible and can be directly processed by a database manager. The representation developed here addresses many of the issues which arise from sharing information among families of software systems.

A central feature of this representation is the handling of system versions. The manner in

which we have defined versions seems to correspond to intuitive notions of version in many cases. Improvements on the definition of version should permit it to encompass other systems relations.

New developments in system representation and the automation of construction will depend on feedback from the software engineering community. Implementations of software construction databases should be developed and applied in real production contexts.

1 Introduction

1. Software Engineering

This chapter presents the author's view of the field of software engineering and the problem of software construction. It is offered a context for the work presented in later chapters but is not directly referenced. A statement of goals and a review of related work appears in Chapter 2. The actual introduction to the technical results of the dissertation is in Chapter 3.

1.1 An Attack on and Defense of Technological Research

The term "software engineering" is a pretentious term which implies that there is an organized collection of techniques for the design and construction of software objects. No such discipline exists, but there are many people contributing to it; perhaps the field will soon deserve its name.

This is a technical thesis, and concerns itself only with the technical details of some aspects of software construction, those commonly referred to as "programming". In a broad sense, this is a mistake. The pervasive technological panacea myth has led some to believe that the problem of software construction will be solved by the construction of good compilers, automatic "verifiers", and flexible run-time environments. Proportionally, too much attention has been given to these issues; poor systems can be constructed using any facilities[Flon75a].

On the contrary, the most important topics in software engineering involve the organization of information and human beings in a manner which is conducive to the production of good systems. Among the issues to be addressed are the managerial decisions, such as the structuring of groups of people, leadership of those groups, information distribution, motivation and responsibility [Tayl11, Wein71]. In addition, the financial and temporal constraints on the resources available to the software project pose a set of crucial problems outside the domain of "programming" and critical to the production of good software. The difficulties of establishing what a system is intended to do, and what is not to do, are often apparent only when the delivered system fails to accomplish its task or, if it does that, fails to be adaptable to the problem which now exists.

Further, there are political and economic questions seldom explicitly resolved about almost all software systems. The reliability of a mass transit system, the feasibility of an anti-ballistic missile system, and the ramifications of electronic funds transfer systems all affect the production of those systems. What responsibility does the implementor of a program have to guarantee that it is appropriately used? How is a software team organized which will construct an information storage and retrieval system in which will be recorded the criminal histories of citizens? For an extreme example, what is the reliability of a piece of a defense system constructed by a draftee?

These are some of the important topics in software engineering. In comparison, the details of the software production process seem of rather mundane scope. Why, indeed, should attention be placed on the technical methods and tools for software construction? There are three reasons for the research in this thesis, plus a confession.

The first reason is an instance of the standard justification for automation, the second regards the state of the computing research, and the third concerns the organization of science research. The confession will follow naturally!

First, the technical problems of software construction are so great at present that they completely occupy the attention of software production personnel. As an analogy, consider the plight of a impoverished citizen: while the poverty is severe, most of the attention of the

citizen is devoted to acquiring the necessities of life, and little is paid to the reasons for his or her poverty. Often it is after the pressure for survival has been slightly alleviated that the citizen is receptive to political programs, union organizers or revolutionaries. Similarly, when the work necessary to merely accomplish the production of a software system swamps the intellectual capabilities of the programming group, the individuals are not excited about the additional constraints under which they should operate in order to make their product maintainable, fault-tolerant, modifiable, verifiable, or otherwise attractive. A dirty system that works brings more satisfaction that a pretty one that does not. Hence, the development of good technical methodologies and tools is a step toward the "liberation" of the programmer. With some of the mental complexity organized, the programmer can pay some attention to more important design, management, and political issues!*

The second reason for doing research on the technical aspects of software engineering is that there is a body of understanding of this topic upon which research can be predicated. The body of knowledge that includes informal prescriptions on programming "style"[Kern74] as well as the more rigorously defined notions of abstract data type[Wulf76, Lisk76], has developed over the last several years, and although there were probably necessary digressions into trivial controversies, there has been also the clarification of the nature of specifications vis-a-vis implementations, for example, and the acceptance of "simple is beautiful" principles in software methodology and tool design. Many of the basic underlying principles are quickly applicable to the design of tools other than programming languages.

The third justification of this line of research is that the present researcher has a significant amount of experience in this field and can contribute more immediately to the state of the art in the technical aspects of programming than to other more difficult realms. This fact is related to the confession: it is easier to do technical research than social, managerial, and political research in technical areas, especially within the context of current specialized academic departments. Technicians and humanitarians are not encouraged by the structure of academia to interact, and the technical expert who knows even the history of her own field (much less her civilization) is rare. The population of historians who comprehend a technological field is similarly sparse. This researcher is committed over the long term to the elimination of the conditions that cause technical research to be done in isolation from humanitarian, political and historical information.

This is, to reiterate, a technical thesis. It contributes to the technical process of constructing and maintaining software. Even the technical scope is limited; only scant attention will be paid to the important technical processes of verification, testing, measurement, and documentation (although the process of documentation is dealt with in parallel to the process of programming). The focus is on the production and maintenance of software systems, with emphasis on the simultaneous existence of several versions of the systems at hand.

1.2 Producing and Maintaining Software

The life of a software product is often arbitrarily divided into two unequal halves. The first half, lasting from the conception of the problem to the day of delivery to the application,

---

*This statement of the value of automation is, unfortunately, naive. It is not generally the case that the automation of an aspect of a person's job results in expansion of the person's scope of concern. In this society, the opposite is the case: either directly, or indirectly, automation results in the narrowing of the attention of an individual worker.[Brav74] Therefore, the claim just made can only be justified in the context of change in the economic system in which programming is done.

is called "development". Great amounts of intellect, managerial skills, computing expertise, computing time, and money are invested, and the first phase of the product's existence is concluded with well deserved congratulatory parties and occasional bonuses. Satisfaction with a job well done is felt by all concerned. All that is left now is -- maintenance.

The designers of the system, who are usually the people with the greatest experience and expertise in computing, probably left the project before delivery; they have taken leave from their current projects long enough to enjoy the festivities. The primary implementors, second in the competence hierarchy, are quickly reassigned to other desperate projects. The rank-and-file programmers, perhaps competent but with limited understanding of the entire process, assume responsibility for fixing errors in the system and adapting the system to the new requirements of the users as they expand the application domain or the role of software systems in it.

This phenomenon reflects the perception that although designing is hard, fixing and modifying is easier. Experience indicates otherwise[Boeh73, Broo75]. In fact, the conclusion being reached by some is that a system must be continuously re-designed during its lifetime or it will descend into a whirlpool caused by repairs that increasingly do more harm than good[Bela71]. Far from being intellectually complete, the "finished" system is the starting point for another difficult task: the coordination of repairs, modifications, redesigns, and growth. The statistics are beginning to show that maintenance is in fact the larger of the two sections of product lifetime, not only in time but in human effort[Boeh73].

1.2.1 Development

Production of software is a complex process. Usually, it is assumed that the initial concept derives from some vague desire to meet a need within the context of an application. Starting with that concept, various transformations occur until a programmed product, perhaps complete with documentation, is delivered and runs [Zelk78, Dund75]. In some cases, these transformations are given names, such as requirements analysis, systems analysis, system design, specification, and coding. There are those who claim that the activities should progress from general concept, through levels of design, until implementation follows as a straightforward extension of a good low level design.

In a good system development environment, the processes of analysis, design and implementation are iterated, as feedback from one aspect of the process impacts decisions in another aspect. Within small development groups, this process occurs naturally and smoothly, since the value of the necessary pauses, reconfigurations, and accomodations is obvious. Even in larger contexts, the value of design iteration is acknowledged, as in Brooks' dictum "plan to throw one away"[Broo75] and Newell's belief that a human "can only design well that which he has designed already"*[Newe77]. Brooks points out, however, that the flexibility of change in the process decreases as the level descends from conceptual design to detailed implementation. This inertia is due to the effort involved in changing individual design decisions after they have been intertwined with others in the lowest level representations of the system.

This inflexibility is a great frustration. Not only are actual modifications difficult and error-prone, but many excellent redesigns of systems (even the necessary ones) are often not implemented due to concerns about cost and reliability. Furthermore, many design improvements are not even developed because the designers are aware of the futility of

---

*This is a special case of the more general statement that a human "can only do well that which he has done before."

proposing radical design overhauls. The development of the first system too often means *de* facto design of all succeeding systems.

The relative immutability of the implementation compared to the design is not unique to software. It is rather the apparent malleability of software that frustrates the sensibilities. It is no surprise that a bridge, once built, cannot be upgraded to carry more weight, since steel beams are expensive and bridges are sequentially assembled over a period of years. But software, it seems, can be changed by an act of will, suitably made manifest. The bits discarded are not wasted; the new ones are no more expensive than the process of specifying them. It is not the cost of the materials that prevent us from re-implementing to improve design, it is our inability to do so with an effort somehow proportional to the "size" of the change.

Nonetheless, systems are developed. That is, the first version is built and delivered[Dund75]. Despite our desires for iterated design, the implementation gets completed (occasionally before the design) and the product "exists". Let us assume, unrealistically, that the design and implementation are separate activities, sequentially ordered in time. What distinguishes design from implementation?

1.2.1.1 Design. Design of a software system follows the establishment of the requirements of the system. It is generally assumed that the task that the system is to perform is reasonably well understood (although see[Parn76a, Habe76] for important limitations to this). However, the task is always too complex to be addressed immediately by programmers. The task must be divided into parts that can be considered in partial isolation from the entire task, and the newly established parts must be mentally recombined to determine if they can in fact interact in such a way as to solve the original problem. And, for apparently robust managerial reasons, the substructure of the system must be such that it can be paralleled by a similar substructure in the human organization that will implement it.* This process repeats, as those parts of the total system are again divided into parts, and those again divided, until the parts are sufficiently small that a human being can comprehend each one completely, albeit individually.

Errors are inevitable in this process. There are errors of many forms, such as misunderstanding the original requirements, dividing along lines that provide little gain in comprehensibility, and hypothesizing parts that are not tractable (e.g. a fast algorithm for the traveling salesman problem). The first division fails perhaps by the third level of design, and a new one is tried. Eventually some design is deemed successful and given the title "The Design". From this instant, redesign is difficult.

What are the criteria by which design decisions (divisions) are made? Generally it is not the case that all the multiply instructions to be used in the system are assigned to a subgroup to implement. Rather, some coherent aspect of the system (from the standpoint of the application) is portioned off. For example, in an operating system, the storage of information on secondary storage is relatively independent from the mechanism for allocating processors to processes; therefore those two aspects of the system will be considered separate subdivisions of the system. In practice, the subparts tend to be functional aspects of the system, parts that "do this" and "do that". Parnas proposed an alternative criterion[Parn72a] and has been acknowledged as the originator of the now popular

---

*This is a somewhat optimistic way of putting this. Normally it is noted that the structure of systems reflect the structure of the organizations that implement them. I am presuming flexibility on the part of the management to respond to the work of the designers.

"information hiding" approach to subdivision (modularization).

1.2.1.2 Implementation. When a given portion of the system has been analyzed and The Design has been determined, the actual process of implementation begins. Some software development groups take pride in completing the design before the coding begins, for this signifies that they have thought through the problems before making poorly conceived attempts to solve them. Although service is paid to the idea of "doing it twice, when possible", the "when possible" admits the underlying assumption that design, once implementation has progressed to a certain point, is not a profitable activity.

There is no justification for subjugating implementation to design. Quality of implementation is, in fact, necessary for realizing the promise of a high quality design. Implementation is not all "coding", if that term is meant to connote the mechanical translation of detailed programming specifications into a given programming language. The implementation of one level of design may require the design of lower level facilities. Hence, the implementation of high level aspects of the design involves almost all the decisions in the system! Some of those decisions, such which data structure to use, may entirely determine the performance characteristics of the system. Two seemingly arbitrary implementation decisions, furthermore, can interact to cause problems, for example, storing a sparse matrix in row order and processing it in column order.

The implementor can be considered to be in the intersection of two sets of constraints. One is the system design we have been discussing; the other is the means by which the implementation is to be accomplished, this provided by the accepted methodologies. The two sets are not coextensive. Programmers often debate whether to accept a solution that is workable but not neat, or a solution that meets the methodological constraints but does not meet well the system goals (e.g. performance).

The implementor of a family of systems must write a family of programs and provide a mechanism for selecting members of the family. Techniques for program family writing must be developed, since single version programs will soon be as unusual as single instruction set computers, single matrix size statistics packages, and single command language terminal interfaces. The focus of effort should be on the development of tools that permit the clear specification of abstractions from programming structures.

1.2.2 Maintenance

The maintenance of software systems is increasingly the major effort of a software project. While painfully obvious to the programmers who find themselves doing more and more maintenance, the software project planners seldom consider as costs the resources that they are obligating to maintenance.

Since human beings all enjoy the process of solving novel problems, it is not surprising that they generally pay more attention to developing the first solution to a problem than to providing for the adaptation of the system to future situations. Among sophisticated software engineers, the latter problem can, however, be considered a problem itself, and some degree of energy is reserved for that task. There is, of course, no support for this expenditure of effort; seldom, in the history of software projects, has the customer walked in to the software group meeting and said, "Please postpone delivery of my system until next quarter so that you can adequately plan for maintenance." And rare is the executive who willingly postpones income from the product today in deference to increased reliability and saleability of the product tomorrow. With the additional financial and scheduling pressures, it is only the

stalwart system designer/implementor who reserves any attention at all to providing, in the design and initial implementation, for the maintainability of the system.

1.2.2.1 Error Repair. Immediately upon the delivery of a software system (and during field testing, if any), the avalanche of error reports descends upon the programming staff. These range from complaints about trivia (e.g. words misspelled in prompt messages) to notices of system disasters (e.g. data bases destroyed).

For some reason, "bugs" and their fixes are considered to be small. Perhaps if we called them fungi or *corpi delicti* their nature and the means for eradicating them would be better perceived. True, some error are "simple" problems of missing statements, inverted signs, and forgotten checks. Most, however, reflect flaws in the design [Boeh75]. Unfortunately, they can sometimes be repaired by a procedure analogous to that used for the repair of simple problems, i.e. insertion of a statement or the check for a special case. But the repair violates design principles and the system has begun its entropic descent. The reported monotonic increase in the number of global variables in a system is a clear illustration of this phenomenon[Bela71].

Why are redesigns not effected when bugs are produced that illuminate the flaws of the current design? First, the people responsible for repairs are not designers, and may not even realize that the error is inherent in the design; their ability to insert a fix insulates them from this shock. Second, the reimplementation is expensive and error prone because tools for recording and reusing implementation information are inadequate. Who wouldn't make the one line fix under these conditions, even though another variable becomes global in the process?

An additional difficulty results from the managerial pressures for quick error repair. The repairs are often needed immediately and cannot wait for the next version of the system, even though there are interactions between the repair and the new features. In addition, the repairs for two different errors may interact or be dependent on each other. Reintegration of repairs with development versions is a notorious problem; it is addressed in part by DeJong[DeJo73].

The existence of bugs cannot be prevented, even by scrupulous care in the preparation of programs; "there is no such thing as a bug-free program, only one in which the bugs have not yet been found." In programming, as in medicine, we can benefit greatly from partially successful efforts at prevention, but we must also provide flexible methods for repair.

1.2.2.2 Enhancement. Along with the error reports come a variety of requests for system improvements. Like error repairs, some enhancements can be made within the design of the original system. Often, however, the enhancements are of a scale comparable to aspects of the original problem, and hence should be considered in the corresponding level of the design. Especially as enhancements accumulate, the incapacity of the original design to comfortably encompass them becomes more apparent.

It must be reiterated that enhancements are usually forced into the old design despite the arguments against doing so. The reasons are the same ones described in previous paragraphs: there is no one available to consider design level problems, and the human reprocessing required to propagate design decisions into implementation details is expensive and unreliable.

1.2.2.3 *Performance Improvement.* One class of enhancements that is particularly likely to result in detriment to the implementation quality of a system concerns performance improvements. The overall goal of the system development team is to get a system that works, albeit slowly or with large storage requirements. Sometimes explicit decisions are made to postpone some performance considerations until the maintenance phase ("when we get some data" or "once we know that it works"). Design decisions significantly impact performance, however, and occasionally the first implementation of a particular modularization or hierarchical structure results in unacceptable overhead. The performance improvements are often made by crossing the boundaries of the design, using the "hidden" information and recognizing the actual limitations under which the current system (but not necessarily later ones) operates.

The above is not intended to imply that performance should or even can be adequately predicted before the system is operational. In many cases, the complexity of the system is such that various measurements must be made before it is known even how the system is expending its resources of time and space. In particular, the interactions of various implementation decisions may produce highly anomalous behavior that can only be understood by a detailed examination of the operation of the system. An entirely appropriate approach to this problem is to construct several versions of the system and compare their behaviors to determine what efficiencies can be obtained. This very process, however, must be envisioned in the high level design, so that variations in the systems to enhance performance are made at the design level and then reflected in the implementation.

1.2.2.4 *Families of Systems.* One particular variety of enhancement is especially important within the context of this thesis. Sometimes the enhancement requested by one user is incompatible with the expectations of another, and a separate but similar system must be constructed to exist simultaneously with the original. (As discussed in section 3.2.1, this construction occurs spontaneously in systems that have development, backup, debugging and instrumented versions.) Most of the information in the two systems is the same, but substantial design or implementation details differ. In [Parn76a, Habe76], this situation is in fact a goal of the design.

The explosion of interconnecting detail in such families either prevents the family from developing altogether, or forces on the programming staff the need for techniques for building members of the family. Examples of the latter are given in[IBM72, Habe76, Erma77, Rein77].

1.2.2.5 *Documentation.* It is possible that the documentation of a given system reasonably reflects the actual design and operation of the initial version of the system. After the maintenance phase for the system is underway, confidence in the correspondence between the system and the documentation becomes weaker and weaker until, in some cases, the documentation is considered more misleading than helpful.

The documentation is usually maintained separately from the system, with some form of communication established between the technicians who design and implement the changes, and the technical writers, who maintain the documentation. This separate maintenance mechanism is unsuitable for the documentation of evolving systems or families of systems. We will consider the documentation to be a parallel implementation of the design of the system. A system and its documentation, then, are enough to establish a system family.

Documents, it should be noted, have an additional property that complicates the problem of

correspondence: they are occasionally printed on paper and distributed over a large geographic area. We will shed no light on the problem of updating paper documents, but will help coordinate the modification of a system and its documentation.

1.2.2.6 The Mechanism of Modification. Once a system is built, modifications are made to the system by changing a system component (perhaps by editing the source program for that component) and by reintegrating that component into a version of the system. For large system, it is not feasible to reconstruct the entire system from source, even with increasingly inexpensive computing capabilities. Typically, some member of the programming team understands the module interconnections and can instruct those who would modify the system in the proper incantations. This process has all the faults of its medieval counterpart. The wizard may cite the spell clearly, but the sorcerer's apprentice may repeat it incorrectly, or apply it to a situation for which it was not appropriate. Also, the wizard may err in developing the incantation, resulting in potentially irreversible deformities in the system. And if the wizard gets hit by a truck, the hapless novices are often left mumbling whatever they can recall that sounds like what the wizard used to say.

Most software projects are forced, after several unpleasant confusions, to adopt some control over the modification of system components and the systems themselves. Usually, some form of "signing out" a program is tried, with or without a mechanized enforcement procedure. However, design level information and construction instructions, not being recorded in single files in the directory, cannot easily be "signed out". For managerial reasons, isolating system decisions in separate files and grouping hierarchically dependent system components in directories is necessary to permit the coordination of people who are simultaneously working on the system.

1.3 Software Methodology Approach

In the terminology of Newell and Simon[Newe72], design is a "search" through a multi-dimensional space; the target of the search is one of those points in the space that corresponds to a viable solution to the problem at hand. For software system design, the space of search is initially constrained only by the nature of the (virtual) machine for which the system is being constructed. Since those machines are often described as "general purpose" computers, operating systems, or programming languages, they are, by definition, designed to be as free as possible of *a priori* constraint. Hence they are only vaguely organized for finding a solution to a given application problem.

Methodologies are principles and techniques for guiding a search through a design space. They direct the search to regions of the space that are probably dense with adequate solutions, although not necessarily to the region containing the best solutions. They consist of general design principles as well as specific knowledge from various levels of the domain of the design. Lessons from experience are accumulated and transmitted in this way from programmer to programmer.

Several fundamental methodologies for program construction have developed in the past several years. Those of importance to this work are discussed below. In addition, new methodologies are in the process of being proposed and explored; some of the more promising of those are also discussed. Other methodologies have come and gone. No attention will be paid to "goto-less programming" or "one page limit" methodologies here. The bankruptcy of such proscriptive approaches becomes obvious when they are scrupulously followed in the text of clearly terrible programs. The nature of a methodology is as a guide, not a law.

1.3.1 Programmer Discipline

Almost all methodologies begin as a set of policies enforced by a programmer on the program structures permitted in programs. This is necessarily the case, since no programming facility can incorporate an unknown policy! The excellent programmers in an installation often develop these policies as a technique for representing some underlying order that they have imposed on the organization of the program.

The surface nature of these policies are often not apparent even to the programmer who developed them. The shoals on which these methods run aground are the beliefs that programming recipes can replace understanding. Even if the programmer does not hold this view, it is likely that the programming manager does.

The appropriate use of programmer discipline is to permit the construction of well organized systems even with primitive or difficult tools. The reiterated claim that excellent systems have been constructed with macro assemblers is well taken; the structure has been defined by the engineering staff and adhered to by the implementation team with the help of the "standards".

At the point that the nature of the methodology enforced by discipline becomes well understood, the general principles that it embodies will be assimilated into a more formal methodology. For example, the discussions of restricted control flow constructs have now abated and the questions are now resolved in each specific instance by balancing the clarity of the program containing the construct with the complexity of the verification rule necessary for its use. This happens not because simplicity of program and ease of verification are both inherently important, but because both are well correlated in experience with reliability and maintainability of programs.

1.3.2 Structured Programming

"Structured programming", perhaps the most abused term in the modern computing literature, derives from the work of Dahl, Dijkstra and Hoare [Dahl72]. Due in part to some unfortunate remarks in Dijkstra's paper [Dijk72], some people came to believe that structured programming was any one of goto-less programming, stepwise refinement, topdown design, or programming in ALGOL-like languages. The essence of structured programming, which is expounded below, was temporarily lost to the literature.

The profound aspects of structured programming concern the use of techniques for reducing the complexity of the programming problem. By emphasizing the "structure" of algorithms, program sections, or data structures, it becomes possible to separate the behavior of the program at one level from the details of each of the components. Hence, for example, it is useful to refine a program in steps because the skeleton of the program can be shown to behave correctly given some properties of the (unexpanded) subprograms. Each of the subprograms can then be considered in turn, in isolation from each other and from the program skeleton in which they are embedded.

Stepwise refinement is an example of a structured programming technique. It is a tool by which a programmer can record one aspect of the complexity of a program in order to direct attention to another aspect. The process of organizing the complexity of the problem is accomplished by the programmer, not by the technique! Other important techniques in structured programming include the definition of abstract data types[Wulf76, Flon75b, Jens74], hierarchical ordering of program segments, and use of

verifiable control structures and operators.

1.3.3 Modularization

Parnas consolidated the informal concepts of modularization [Parn72a]. Other concepts, such as separate compilation, are not now included under this term. The primary attribute of a module is that it hides a unit of information. This may be a small implementation detail, such as the actual location of a logical device, or a major design decision, as in whether to sort a KWIC index before printing it or at the same time[Parn72b].

Modules contain and hide information. Sometimes they can be encoded as programs or program segments, but not always. For example, the method for ensuring data integrity over procedure calls (the "calling sequence") can be contained in a module, but the actual method may not be the same for two different caller/callee pairs. The syntax of a command language, for another example, is independent of the means by which it is represented in a program.

The recent exploration of abstract data types is an important contribution to the process of modularization. For many purposes, a module can take the form of a data representation combined with a set of permissible operations on that representation. Properly described, it serves as an implementation of a mathematical object that can be discussed separate from its implementation. This permits the abstract behavior to be made available to a program that used the data type while hiding the mechanisms by which those behaviors are accomplished. This form of modularization is so widely applicable that it has become the primary emphasis in the design of several new programming languages [Lisk76, Wulf76, Wirt77, Lamp77, DoD77, Gesc77, Ambl77].

The process of modularization results in fragmentation of the system representation. When systems were monolithic expanses of source code, recompiling the system was all that was needed to integrate the system into a runnable form. The problem of systematically combining modules into systems is basically unsolved, although the Mesa designers have had some success[Gesc77].

1.3.4 Hierarchical Structure

The division of systems into levels is most often attributed to Dijkstra[Dijk68]. Most of the experience in this field has been provided by operating system projects[Lisk72, Habe76, Saxe76, Neum74]. Parnas[Parn76a] and others have argued for such organization of other systems.

A hierarchically organized system has an enforced partial ordering on some objects in the system. Such a system may be described as a sequence of levels, each level being defined as the class of objects that are inferior (according to the relation) to objects in higher levels and superior to those in the lower levels.

Careful selection of objects and relations can result in a system with some very nice properties. By using "processes" as the objects, and "provides work to" as the relation, freedom from deadlock in the system follows immediately[Dijk68]. If "functions" are restricted by "calls" relations, so that function calls always descend a level in the system, stack depth can be limited to a known maximum. And, if sets of functions are ordered by "functional dependency"[Habe76], each layer corresponds to a virtual machine that can be programmed without dependence on the upper levels.

Hierarchy in a system is often established within the information hiding lines. Although this has the advantage of simplicity, difficulty arises in real systems either with establishing a hierarchy at all, or in efficiently implementing the system along the hierarchy established. Since the two concepts are not dependent, hierarchy can be established "orthogonally" to modularization, and the benefits of both structures will obtain in the resulting system[Habe76].

The clear specification of the hierarchical relation is necessary to a clear understanding of the system structure that will result[Parn74].

1.3.5 Language Design

A natural development in the consolidation of a methodology is that it be incorporated in a notation. In the case of programming methodologies, it is usually assumed that this notation will be processable by a program, such as a compiler (although see[Schw79] for a different use of notations). The abstract data type methodology is currently in this phase of development, and several languages are being designed around the concept.

Generally extensible languages, in which a variety of methodological constructs might be developed, do not serve the same value as languages that incorporate specific methodologies. The nature of a methodology is both inclusive and exclusive, so that a language that permits the prescribed structures and facilities must also prevent or discourage the proscribed ones. Furthermore, for a compiler to be constructed that both is efficient and produces efficient programs, the information about the probable use of the language must be available. Hence, Alphard[Wulf76], CLU[Lisk76], and Mesa[Gesc77], as well as others, have the abstract data type concept integrated into the language and compiler design.

Not all software methodologists believe that this is a necessary or even appropriate step for a methodology to take[Parn77]. Any rigidity in the programming tools around a specific methodology results in difficulty in constructing programs or systems that either do not fit into the methodological framework or need extensions of the methodology not supported by the tools.

Our view is that languages are an important stage of methodological development. The primary reason is that languages both define and confine the realm of discourse, and such restriction is valuable to human beings in organizing complexity. Secondarily, constraint on the possibilities makes automation of various aspects feasible, and that is one of the tasks before us.

1.3.6 Verification

A technique for improving software quality that is popular in academic circles is the application of mathematical proof techniques to the behavior of programs. Often mistakenly called "proof of correctness", the actual achievement is a proof that a semantic interpretation of the program has certain properties in a abstract model of the domain of the program. The correspondence of that model to the actual domain (e.g. range of integers) must be established ad hoc. More profoundly, whether that model and the now guaranteed properties correspond to the desired behavior of the program is a matter for deep meditation and frantic handwaving.

Program verification is an important conceptual breakthrough. For certain critical system components, and and those with particularly tractable mathematical properties, there will be

no substitute for exploitation of mathematical techniques. Recent developments in formal verification have shown that mathematical rigor is not restricted to greatest common divisor programs, and that assertions can be proven that have a close connection to the expectations we have of programs [Flon77].

It is also true that almost all program code will not be processed by the long awaited automatic verifiers. Large amount of system code are concerned with information processing that is not mathematically tractable. The effort that would be put into constructing a mathematical model of the real situation, if applied instead to the program itself, would result in a much more trustworthy conviction that the program would behave as desired.

We will contribute to the validation of systems by recording construction procedures. As elucidated in[Flon77], one can establish a variety of useful properties of a system if one builds the system with components that cannot invalidate those properties. For example, suppose the operations on processes provided as part of an operating system are of the form "move process from list1 to list2". It can easily be established that if processes in the system are initially in one and only one list, then every process will always be on one and only one list. This form of verification of properties of systems, i.e. designing such that invalidating the property is impossible, has its analog in system construction. We will propose, for example, that the operations on language procedures be of the form "get procedure specifications" and "get procedure definition", so that it will be impossible to construct an external reference to a procedure that is not compatible with the actual precompiled procedure body.

1.4 Software Technology Approach

With the construction of the first symbolic assemblers, the technological attack on system construction was initiated. In the intervening twenty years, a wide variety of programs have been developed with the primary function of assisting programmers to construct other programs.

1.4.1 General vs. Specialized Tools

Each software system project generates a collection of tools useful in that project[Kern76]. Depending on a variety of factors, such as programmer prescience and availability of time, the tools may be useful to other projects either as they are or with modification. Occasionally, some resources become available to generalize the work, and tools of wide applicability are generated.

The evolution of tools from specific to general occurs in parallel with the continuous addition of features intended to solve specific problems encountered in the work of the programmers. Occasionally, the ad hoc addition of features swamps the trend toward generality, and a software tool stagnates or falls into disuse. Enhancement of tools is different from enhancement of other systems only in that, because they are "in house" products, reliability and maintainability are often given even lower priority than for customer systems.

The tradeoffs among generality, simplicity, and applicability are particularly acute in software tool construction. Historically, software tools have been either overrestrictive in the capabilities provided in exchange for clarity and simplicity of the tool and its product (e.g. Pascal) or so over-general that the features of the tool interact dangerously and the production of efficient products is difficult (e.g. PL/1). Only partial success has been achieved in moderating over-general tools, such as in the command procedure libraries for common job

control language sequences. And providing features in an overrestricted system, if possible, is usually done at the expense of some of the usefulness of the underlying system, as when directory systems are built on top of a basically inadequate file system.

It is perhaps inevitable that, before the working domain of a software tool is well understood, the pendulum must swing toward generality until relationships among elements of the domain can be explored, and toward simplicity until the limits of the model of the domain can be delineated. Over a large scope, design must be iterated and the knowledge obtained by using tools cycled back to the designers.

1.4.2 Abstraction-providing Tools vs. Transparency

The builder of a software tool attempts to provide some facility not immediately available in the environment. Often this facility abstracts in a useful way from the details necessary to accomplish a task. For example, a compiler abstracts from the instruction set and from specific memory locations by providing such constructs as expressions, procedures, records, and data types. Usually some capability available in the underlying system is no longer available in the abstract environment [Parn72c].

The removal of capability requires justification of the most convincing form. The elimination of arbitrary transfers in programming languages, for example, must be substantiated by arguing that the remaining language is "adequate" in various senses for all reasonable programs, that significant gains can be made in analysis of the programs so constructed, and that no real cost has accrued.

Even the lowest level tools tend to make otherwise simple operations become difficult. Programming languages typically remove from the domain of the programmer the hardware stack, the registers and the procedure invocation mechanisms. Implementation of features that require control of these resources are accomplished, if at all, by subverting the compiler or exercising a "descent to machine code" operation and observing the *caveat implementor* sign that is attached.

Perhaps the most common case of capability loss occurs in the operating system interface of compilers. In almost all languages that provide I/O facilities, a large proportion of the I/O capability of the operating system cannot be exploited by user programs. The number of high level languages that allow use of the job control block, the communication buffer, or the contents of directories is exceedingly small. These restrictions either prevent use of the tool for these applications or cause subversions.

The designer of a programming language, or other abstraction providing tool, should explicitly record those capabilities of the underlying system that will not be available to users of the tool. While it is clear that the general registers can not be manipulated by the user of a programming language, one must warn a user that synchronization primitives, interprocess communication, direct access disk operations, real-time clock requests, and so forth, are beyond the reach of programs in the language. Not only does this alert the user to potential limitations on the applicability of the tool, but helps the designer to understand the ramifications of providing features that depend on restricting the user.

1.4.3 Compatibility of Tools

Software tools are much more useful if they are compatible with each other. For example, a compiler that produces a symbol table in the format used by the interactive debugger for another language can make feasible the use of that debugger for its object programs, and

facilitate the creation of a general interactive debugger for the languages available in an installation.

Compatibility of tools is achieved routinely with object code formats that are defined by the linking loader. Command processors, editors, macro processors, documentation programs and compilers seldom share even rudimentary concepts, such as the definition of a lexeme. Accomplishing integration without removing important flexibility from the individual tools may require the establishment of a total programming support environment in which the design problems of the entire tool collection are considered.

1.4.4 Abstracting from System Components

Programmers abstract easily from the domain of system. Told that the domain is airplanes, a programmer will quickly (perhaps *too* quickly) develop images of the required abstract features of airplanes in the context of the system. If it is an airline reservation system, "airplane" means as seating arrangements, flight equipment, food service requirements, and so forth. If it is an airplane maintenance system, "airplane" consists of parts, subsystems, service histories, etc.

On the other hand, only a few programmers effectively abstract from the tools that they are using to abstract from the domain of the system. Presented with an algorithm for constructing a part of a program, it is likely that the programmer will *execute* the algorithm rather than *encode* the algorithm for another program to execute. Suppose that a programmer is given an algorithm for describing the seating arrangement of an airplane: "for each aircraft chart, create a list of rows, where each row has a number, a smoking/no-smoking indication, a class, and a list of seats, and where each seat has a letter and an indication of whether it is a window, aisle or internal seat". Rather than reflect upon her own actions, the programmer will likely sit at a desk, keypunch, or terminal, and write the code for the specified structures.

Some programmers object to spending their time in such activities, and note that changes to airplane descriptions (or the addition of new airplanes) will result in changes in the program that represents them, rather than to an airplane description notation. Using the macro processor of the programming language, they will attempt to develop some program-like mechanism for converting an airplane representation to the desired programming language constructs.

Tools for the task of abstracting from programs have generally been poorly engineered. Prior efforts attempted to provide these tools within programming languages. The result is often an inordinate amount of attention paid to evaluation order, internal compiler processes and distinctions between linguistic types (e.g. the definition of a lexeme, the difference between a character string and the characters in the language that represent a character string constant containing the same characters).

1.5 Theory vs. practice

All knowledge comes from interaction with the natural world[Mao48]. Theory, as the development of an organized model of features of that world, serves to provide structure for knowledge and to guide the search for new knowledge. Theory follows from, and does not precede, experience.

This pattern applies to any scientific field, but is particularly relevant toward guiding research in engineering disciplines. The collected experience of the thousands of

programmers is the source of the information from which a theory of software engineering can be distilled. The criterion for success of an effort in software engineering must eventually be the applicability of the result to the construction of software products. This cycle insists that software engineering theorists must interact with software practitioners, and the best understanding can be achieved by theorists who are being involved with real software projects.

The extension of knowledge by theoretical exploration is occasionally productive. One step in that direction usually suffices to exhaust the actual information supply, however, while two or three are indication that large amounts of vacuous research is being attempted. For example, further exploration of the mathematical qualities of simple programs is unlikely to produce significant improvement system reliability, modifiability, or performance at the present time.

It is also necessary that academicians not limit their scope to software projects in academic environments. The problems of the software engineer are often masked by the flexibility of the user community, the fuzziness of the accounting mechanisms, and the uniqueness of the implementation. A typical phenomenon: documentation of systems in academic environments is often fragmentary or non-existent, and folklore plays an important documentary role in these isolated subcultures.

1.6 Introductory Conclusions

The central theme of several current methodological movements in software engineering is the organization of information. Under such terms as "abstraction", "hierarchy" and "structuring", these methodologies emphasize techniques for expressing, encapsulating, and separating aspects of the complexity of systems.

As a natural development of a methodological movement, tools have been developed that incorporate the concepts and techniques of that methodology. The progress in programming language research exemplifies this process with respect to abstraction.

The system integration problem is a result of the decomposition of systems into modules, levels, data types, and so forth. The techniques for describing programs have not been applicable for describing how modules are interconnected, how different versions of module are related, how systems are constructed and how system families share information.

There is a present need for a technique for describing the integration of systems. The benefits of modularization can be fully achieved only if the process of constructing systems from modules can also be clearly expressed. The evolution of tools supporting these techniques will ease the process of developing and maintaining systems. In addition, the implementation process can be partly recast as a construction process. The same tools will reduce cost of reimplementation and thereby reduce the cost and risk associated with the design/implement loop.

Programming languages contribute to system reliability, maintainability and flexibility by improving the way in which program information is expressed. The development of equally appropriate system integration languages (and language processors) will produce equally positive results. And by correctly addressing simultaneous maintenance of system families, a new dimension of flexibility and engineer productivity can be achieved.

The most reliable basis on which to develop techniques for system integration is the experience of veteran software engineers. This work is an attempt to collect some of those

experiences into a coherent framework. At some time, it will be reasonable to develop a theory of software system descriptions as has been done with programming language syntax and semantics. At the present time, however, more information must be collected about the processes of software system construction.

2. Goals for the Thesis

The primary goal of this thesis is to develop a representation for families of software systems. In support of that representation, a rudimentary system has been constructed which lends credence to the claim that the representation is useful. To illustrate the efficacy of those efforts, we have applied the techniques to some aspects of a real system that is known to contain some difficult system construction aspects.

The particular representation developed here is an intermediate between an ideal user interface and an internal data structure. This selection will provide us with some insight into what is required in both directions for future systems.

2.1 Goals for the Thesis Elaborated

The selection of a representation for a system is determined by the purpose that the representation is to serve. The primary function of this representation is to facilitate the iteration between design and implementation and to assist in the maintenance of multiple coexisting system versions. It will take some effort to bridge the gap between the design of an entire system family and the specification of the construction of the lowest level detail of the implementation.

2.1.1 Integration of Design and Construction Descriptions

2.1.1.1 View from the Bottom. The representation that is developed here must not only permit the automation of the system operations necessary to the actual construction of the systems at hand (e.g. compilation, editing), but must also be organized in such a way that it elucidates the structure of the system along lines meaningful to the designers and programmers of the system. Compilation of a program, for example, results not from a command which directly invokes a compiler, but rather from a command which builds a particular version of a particular system for which that compilation is necessary.

What is a reasonable view of the processes involved in the construction of software systems? What restrictions on implementation form can we propose and which must be left unconstrained? From the transparency arguments in 1.4.2, we must not restrict implementors unless we can provide important advantages that can only be obtained by such restrictions.

- **Languages.** The choice of implementation language is a difficult and important aspect of the implementer's task. To preempt any aspect of this choice is to limit the variety of implementations that can be considered for each individual task. There is no justification even to restrict the implementor to some one language (although there are advantages to working in a single language which should be considered). The case is obvious for systems that operate on two different hardware configurations simultaneously. In section 6.1 we see an example in which a common high level language for a general purpose computer must interact with a program for a mini-computer constructed in assembler language. Yet another example of multiple languages is forced by the interaction between the system and its documentation; it is quite unlikely that there will exist a language for which compilers exist for generating machine code and also printable documents!

- **Target Machines.** Many systems are intended to run on the same computer and operating system as that on which they were constructed. Others, however, are

not, especially those constructed for small computers with little or no sophisticated software construction apparatus. Therefore, we will construct systems without necessarily having control over the execution environment of the system nor the ability to obtain information from that environment.

- **Installation Specific Data.** There is a large collection of information that is specific to a given installation which is used for system construction. Accounting procedures, job control languages, and file system formats must be exploitable by programmers, although no uniform methods for abstracting these details exist. We isolate such aspects from the processes inherent in system construction without preventing intelligent use of the operating system resources.

2.1.1.2 View from the Top. Conversely, the representation must clearly portray a significant aspect of the design of the system, while providing the capability of specifying the most mundane details of how that design decision is actually carried out. High level design languages that are not integrated with the actual construction process are useful to designers, but do not provide the flexibility necessary to iterate design or maintain system families.

- **Design Languages.** The specification of notation suitable for recording some of the design decisions in a system is one of the explicit tasks established for this thesis. This is not intended to exclude the use of complementary languages and systems for other dimensions of the design. Other researchers and software practitioners are actively exploring design languages and tools [Ross77, Teic77, Carp75]. We will include at least those aspects of the design necessary to control the software construction processes.

- **The Nature of Resources.** The notion of module or subsystem has become familiar, but the methods for interconnecting modules are not well understood. It is fairly common to conceive of a subsystem as "providing" a set of resources to users of the module, but there is no consensus about the attributes of "resources". We will permit resources to represent any facility that the system designer can represent as a character string. Other information must be converted to character string form by special tools.*

2.1.2 Hierarchy of Design

The implementation of one level of design requires the design of the next lower level. Despite the implications of a previous section, design and implementation alternate in a hierarchy parallel to that of the system itself. The representation must reflect this alternation by permitting the designer to specify at any point in the system only that information appropriate to the particular context. Hence, the relationships within the information necessary to construct subsystems of a system must be invisible to the representation of the higher level system, and construction details for subsystems must likewise be hidden from the construction of systems that use them.

---

*The term "resource" does not apply, in this context, to the consumable substances traditionally allocated by operating systems, e.g. storage space and computation.

Suppose that program PrintCustomerList will use the operation ApplyFunctionToListElements (MAPC in a LISP environment) to invoke the PrintCustomer operation on each of the elements in the customer list. The PrintCustomerList program will be constructed in some manner determined by its own structure. The method by which the ApplyFunctionToListElements operation is made available to this program is not relevant to it.

In the context of the ApplyFunctionToListElements program, however, the choice between providing a set of macro definitions or a set of references to external routines must be made, and the origin of those routines must be specified. All of this information should be as independent as possible of the potential users (such as PrintCustomerList) and of the mechanisms necessary to construct those users.

It is not possible to completely isolate these dependencies, of course. If PrintCustomerList and ApplyFunctionToListElements are written in incompatible languages, there may be no way to interconnect the two. Furthermore, linguistic constraints may make it necessary for a subprogram to know something of the context in that it will be compiled. The first problem we address in this thesis, while the second will wait for the results of other researchers [Tich80, Schw79].

2.1.3 Families of Systems

In section 1.2.2.4 the notion of a system family was introduced. The concepts and system developed in this thesis are primarily motivated by the difficulty of maintaining multiple versions of systems in a consistent state. It is not sufficient, however, to duplicate information in order to duplicate systems. The implementor of a system family must be able to maintain a module or subsystem as a single unit, regardless of the different systems in that it is incorporated.

Suppose that a graphics plotting subsystem is incorporated in an executive planning system. The plotting subsystem makes use of lower level subsystems such as a command processor, graphics line and curve drawing routines, mathematical subroutines, and so forth. The plotting subsystem, in fact, exists in two versions, one for a raster scan display and another for a vector display (we will ignore for the moment the possibility of alternate versions of the low level programs).

The specification of the system structure for the executive planning systems will include subsystem interconnections and construction information relevant to building the system at that level. Included will be an indication that the plotting subsystem is required. The system structure for the plotting package will likewise specify the organization of subsystems necessary for plotting to be done; that structuring is, we hypothesize, independent of the raster or vector orientation.

The mechanisms for construction of the planning systems that incorporate the plotting packages must be such that changing either the algorithm for a particular plotting function or the relationship of subsystems within the plotting subsystem is registered in the two executive planning systems. Providing a single system representation that can be copied to form "families" fails to accomplish this goal. The common implementation knowledge shared by members of a family must be encapsulated within the context of the system representation.

2.1.4 Exclusions

The following areas are not directly address in this thesis and these descriptions are boundaries on the thesis scope.

- **Environmental Systems.** Systems such as APL, LISP, and L* incorporate their own system building mechanisms. By preempting the editing and interconnection aspects of system construction, these systems are left insufficiently flexible to integrate into a unified system structuring system.

- **Algorithmic Knowledge.** Some of the knowledge about systems consists of algorithms and data structures that comprise the system. We will use specific algorithms or data structures only as examples of the kind of system components that could be connected together to form a software system.

- **Programmer Knowledge.** A model of the programming process or of programmer knowledge. We attempt to automate some programming procedures but not to attempt "automatic programming."

- **Environment Specific Techniques.** The concepts provided in this thesis are hopefully general and independent of particular languages, compilers, document generators, and so forth. Therefore, details of specific techniques applicable to specific processors will be avoided except when used as illustrations of how a general facility can be tailored to a particular processor.

- **Verification of Component Specifications.** Certification that format specifications of a subsystem meet the requirements of its users. Other researchers are currently examining this aspect of the problem [Tich80].

- **Type Checking.** A specific form of specification matching is often performed during linking of system components [Gesc77, Wulf76, Lisk74a], namely, checking that the types of arguments and parameters are compatible. Since this analysis is completely dependent on linguistic features of particular programming languages, the programming language facilities will be invoked to perform such checking.

- **Text Editing/Macro Expanding Capabilities.** A good text processing facility, text editor and/or macro expander is crucial to high productivity of programmers and to the implementation of many of the concepts advanced in the thesis. We assume that we have such facilities available.

- **File System Capabilities.** A good directory or catalog combined with a flexible file representation can contribute directly to increased programmer capability. Good ideas in this area are described in [Alme77, Roch74, Orga, Rich74].

- **Access Control to Files.** Much of the control of system evolution can be accomplished through restricting and recording access to the files that represent the components of the system [Habe77]. In particular, file histories provide a rather good approximation to program histories.

- **Integration Techniques.** Once the abstract modularization of a system has been accomplished, the task of representing each of the modules begins. Much of the

concern over this representation hinges on the mechanisms by which modules will be recombined to form entire systems. Some of the examples will make use of particular methods of editing text, combining program segments, linkage editing, and so forth, in order to illustrate the application of our approach to conventional integration techniques.

- Dynamic Modification. In order for a system to be modified during its execution, special hardware and software facilities must be established. Such mechanisms are independent of the structure of the systems which might utilize them.

2.2 Review of related work

Very little research and only a modest amount of engineering has been devoted to the topics explored in this thesis. The following sections review several efforts, some directly related and others philosophically sympathetic.

2.2.1 Krutar Flexors

Rudy Krutar has explored some unusual types of system modifications and proposed some implementation mechanisms[Krut75]. The particular examples he discusses are extracted from his experience with a variety of software systems. He represents the various versions of the systems as data flow graphs with coroutine nodes; a node can be replaced with another node or a subgraph to implement a modification.

The particular "flexors" he develops (a character set redefinition facility, a method of defining programming language semantics, a method for creating lexical analyzers, etc.) are of interest in this context as examples of portions of systems which are usually cast in concrete. By providing such flexors as library facilities, construction of complex systems which must have, for example, a lexical analyzer, becomes more economical.

Krutar intuitions are similar to those of the present author. His implementation techniques are one approach to achieving the flexibility that is his primary objective. We will attempt to represent statically the flexibility he achieves dynamically with coroutines.

2.2.2 Parnas' System Families

Dave Parnas introduced the "software system family" term and justifies the consideration of system family design[Parn76a, Parn76b]. He argues that similar system can be designed together, so that the facilities which they all require can be designed and implemented once (or a small number of times) rather than repeatedly within the context of each system.

Family design is facilitated by rigorous application of some particular design methodologies, namely modular decomposition and hierarchical design with a well defined hierarchical relation. The independent application of those two methodologies results in a rich set of alternative systems. Some of these concepts were adapted to design a family of operating systems[Habe76].

2.2.3 Software Factory

At System Development Corporation, Bratman and Court have developed a software development database with a integrated collection of tools for manipulating it, collectively

called the Software Factory[Brat75]. The database includes not only system integration information but several types of management information, such as schedules, test results, development status and specifications.

The factory has a common command language and several processors which operate on the database. The command language system also accumulates statistics about the development process, such as computer resource usage, test run results, and module sizes.

A management tool produces information about schedules, milestones, component status, and resource dependencies; it incorporates management techniques such as PERT. By incorporating some system design information in the management part of the database, the reports and schedules are organized parallel to the software design structures. Thus the impact of a design modification can be evaluated in terms of the management resources implied by the design change.

More technical facilities are provided by the engineering tools which operate on the program representation part of the database. A documentation tool generates program documentation from specially formatted comments in the program text. A program flow analysis and measurement tool automatically inserts measuring hooks into programs that can produce execution profiles. A test case generator automates the process of exercising all program statements. And a top-down design tool allows partially implemented programs to be executed with program stubs in place of actual components.

Integration of the management and technical aspects of software development in a single database system provides considerable leverage on the control of software development. The authors have paid particular attention to the management control of development although some of the management techniques parallel the technical techniques. The database system proposed in this thesis would be an extension of the technical portion of the software factory database; it could presumably be integrated with the other dimensions.

2.2.4 Boeing Software Design Validation Tool

The Software Design Validation Tool (SDVT) developed at Boeing Computer Systems [Carp75] is a partially automated design representation tool. A system is functionally decomposed into a tree, that is, a set of functions is decomposed into a set of more elementary functions. Each level in the tree is augmented by a transition diagram that explains how control might flow between the various program components at that level. Modification of global data is represented by noting within the transition diagram the type of operations that might be performed on each "data parcel"; a parcel can be created, read, modified or destroyed.

This representation of the system can be processed to determine if the system is consistent according to such criteria as "can it terminate?", "is every design state reachable?" and "are there collections of states with no exit?" A system, called DECA, processes the descriptions and matches the specifications of system components.

Like the system developed in this thesis, DECA processes a design language. It is typical of commercial design languages in that it concentrates on control flow aspects of a program rather than information distribution aspects. While there is a real need for this type of tool, it would be an adjunct to a system of the type proposed here.

The SDVT notation is not connected in any way to the process of system construction. Therefore, the design goals established by the notation are implemented in the conventional

manner, with programmer discipline and management intervention.

2.2.5 DeJong System Building System

The System Building System developed at IBM Research [DeJo73] is a system for the automated construction of systems. Systems are viewed as executable collections of files generated from source files, represented in PL/I or IBM assembler language. In fact, the system parses PL/I and assembler programs to determine what resources they require. SBS is implemented as an extension to CP/CMS. Under user control, the system performs system compilation and linking operations. Information about the system structure is maintained in a relational data base that is updated by SBS when construction steps are completed.

An interesting feature of SBS is its attention to multiple versions of systems. Each file and system is assumed to exist in several versions, many of which are active simultaneously. The versions of files are numbered, so simultaneous existence of versions of files is a temporary condition; at user designated times, divergent versions of files are "synchronized" into a single completely updated file. System versions are organized into trees, with the operational version at the top, and various development trees appended to it; at user specified times, systems are "promoted" from a lower development status to a higher one and the changes to the various subtrees are combined by "synchronizing" the files that comprise them.

Updates to files that are maintained by SBS causes automatic invalidation of system link files that depend on the changed files. There are mechanisms for preventing or delaying these modifications, and for controlling the combination of divergent changes to files.

The definition of higher level construction operations is similar to the proposal put forth here, as is the central data base from which information about construction is extracted. This system deserves special credit for addressing the problem of multiple versions, although only a restricted concept of multiple version is considered.

The weaknesses of SBS include its PL/I specific implementation and the narrow view of system construction that derives from considering source programs in a particular language as the objects from which systems are constructed. The command language, and other aspects of the system, are quite crude and unnecessarily restrictive. The derivation of system structure from the actual references that modules make on each other is inferior to the explicit designation of module interconnections.

The worst problems of the system, however, stem from its strengths. In attempting to solve some multiple version problems, a "production" model of a system was assumed; that is, each version of a system is a part of the development of the single "operational" version of the system. Only with this model do the concepts of "synchronization" and "promotion" make any sense. Furthermore, it is difficult in SBS to share information between independent system trees, especially if that information exists in several versions as well. Also, the SBS data base does not naturally facilitate the representation of hierarchical system design; all objects within the data base are at the same level in all respects.

2.2.6 Habermann System Design and Maintenance Control System

In the SDC system implemented by Habermann[Habe77], the notion of "typed object" is extended to the files that comprise the modules of a system. A system is composed of modules each of which is an abstract data type, or a collection of abstract data types with additional facilities that relate the types in the collection. Programs implementing the modules

reside in files that are organized in a tree structured directory. A protection mechanism regulates access to the files.

The files of a module are organized into two groups: the "basic" files, original source information or history, and "working" files, produced from the basic files. Operations on a module result from user commands such as "update source", "print history log" or "copy source file". The system enforces modification policies by checking the commands against an authorization list provided by a manager. Maintenance of backup copies of modified files, definition of testing environments, some module interconnection specifications (within module definitions) and automatic invalidation of working files that are out of date are also provided by SDC.

This system, and the one proposed in this thesis, share the idea of a central system construction data base with high level construction operators. In addition, the auxiliary files that are a part of a module in this system reappear in our system as "histories" and "mailboxes", each with a set of operations defined on them.

The system is limited in that the primitive "module" in the system is assumed to be a programming language object, and we consider that model insufficient for a general module implementation. The descriptions of entire systems do not appear in the SDC context. We will separate the module interconnection information from the module definition both for philosophical reasons and to make possible the definition of multiple versions of systems, an issue not addressed in the Habermann system. We will also pay more attention to the automatic generation of system construction activities.

2.2.7 DeRemer and Kron MIL

An immediate precursor to the concepts explored in this thesis is the module interconnection language (MIL) proposal of DeRemer and Kron [DeRe76]. They argue convincingly about the difference between programming-in-the-small, for which a programming language is appropriate, and programming-in-the-large, for which a module interconnection language is needed.

The MIL that DeRemer proposed is oriented around a tree model of system composition. Each system is divided into subsystems, each of which might likewise be subdivided. Resources are explicitly routed from the "provider" of a resource to the subsystems that "require" them. A subsystem can "provide" a resource by "originating" it or by "deriving" it from an offspring system. Access to resources supplied by siblings is controlled by the parent system. Resources available to a system are usually available to the offspring systems, but resources do not bubble up unless specified to do so. Programs are attached to subsystems as the mechanism for "originating" resources. (The resources are considered to be functions, or other programming constructs.)

The MIL "programs" are submitted, hypothetically, to an MIL "compiler" that produces completed systems. The MIL compiler checks module boundaries for compatibility, verifies that resources are used only where they are available according to the MIL program, and displays actual resource usage patterns (eliminating the multistep access path).

This thesis extends the notion of the MIL to include several other concepts necessary for multiversion systems constructed from fully general modules. The existence of a processor for the MIL was also derived in part from DeRemer's suggestion. However, the notion of an MIL compiler has been discarded in favor of a system construction data base processor supporting an interactive system construction environment. The restrictive assumption about

the contents of program files, and their attachment at specific node sites, has been eased, and the specification of construction procedures has been made explicit.

2.2.8 Thomas MIL

Thomas also developed a module interconnection notation and discussed a possible module interconnection processor [Thom76]. His approach was to build a module interconnection language (MIL) that defined "environments" of accessible names; some of these names refer to resources while others denote modules, subsystems, or nodes in the MIL. The resources are assumed to be CLU-like resources (i.e. clusters or procedures) that are, eventually, provided by source code modules. Each module provides a set of resources and may require a set as well. Modules can be combined into subsystems that act (almost) like modules within the MIL.

A node in the MIL describes an environment, or set of accessible resources, and a group of modules that will share that environment. If resources are used by those modules, they are either acquired from "successors" of this node, specified within it, or from the node that is using this node. (The latter alternative is the source of a great deal of complexity and difficulty in the implementation of this MIL system.)

Thomas also discusses a system that performs the interconnections described by the MIL. To him, the most cogent issue is whether to determine which actual module is to be used for a given resource at compile time or at link time. Since Thomas is bound to the compile/link paradigm, those are the only real alternatives available. Within this thesis we will explore many different binding schemes and not be limited to the conventions of any one compilation system.

2.2.9 Tichy MIL

In related work at Carnegie-Mellon University, Tichy is developing a module interconnection language and exploring several issues relating to system construction[Tich80]. The three primary foci of the module interconnection language, the interface between the MIL and the compilers, and the organization of asynchronously evolving multi-version systems.

The MIL itself is similar to those of Thomas, DeRemer and this thesis. The resources are constrained to be programming language objects such as data type definitions or procedures, and the interfaces are specified within the module interconnection language. Part of the responsibility of the MIL processor in this system is module boundary type checking.

It is planned that portions of the systems can be compiled separately in any order, but that complete interface checking will be provided. In order to accomplish this, the MIL processor must have additional information available from compilers. The C compiler is being modified to test the feasibility of asynchronous compilation.

Tichy's project complements the current work by exploring how processors might be designed to make system descriptions more convenient and flexible, and by developing MIL strategies for organizing the asynchronous development of parts of a system.

2.2.10 Clear/Caster

The "Controlled Library Environment and Resources" system and the "Computer Assisted System for Total Reduction of Effort" system were developed as internal tools for IBM development projects [Brow70]. Clear maintained a database of objects containing programs,

macros, specifications, documentation, and messages. A major motivation for this database was the control of the development process from a management viewpoint. Caster extended Clear with a terminal-oriented interface that provided interactive access and modification of the database, remote job entry and conversational debugging.

Included in the Clear/Caster context are several special facilities: Multi-version files can be stored with "deltas", cross-references are extracted, program libraries are maintained, a document preparation program facilitates manual writing and others.

Clear/Caster is an old system and exploits none of the recent methodological advances. It is unclear that a new system would incorporate any of the detailed structures of Clear/Caster, although the underlying principles are sound.

2.2.11 Programmer's Workbench

The Bell Labs Programmer's Workbench (PWB) is described in [Dolo76a, Mash76a, Mash76b, Dolo76b, Knud76, Bian76, Roch74]. It is an entire computing system dedicated to the process of preparing programs and documents for system that run on other computers connected to the PWB machines via hardware communication lines.

The authors argue convincingly that the facilities needed by program developers are dissimilar from those required by program users. In particular, the large scale equipment, complex operating and file systems, protection mechanisms, and "compatible" architectures that are crucial to the operation of systems are contrasted with the small, flexible, interactive environment ideal for program development. For example, systems are often constructed to handle massive amount of data and therefore the data base facilities of the host for such a program should be sufficiently rich to permit maximum exploitation of the hardware. Program development data bases, however, are typically small, and the benefits of a simple, uniform data base facility in terms of the ease of constructing new tools to operate on it, far outweigh the inefficiencies that may be inherent in it.

The facilities currently supported by the PWB are a source code control system, a remote job entry system, a document preparation system, a modification request control system, and drivers that simulate user conditions for testing. The source code control system is a file storage system that records, in the same operating system "file", the various versions of a text file; this is accomplished by recording the original version plus interleaved modification descriptions ("deltas") that can be applied to create more up-to-date versions. The remote job entry system provides the interface between the PWB system and the host systems on which compilation and system testing are carried out. The document preparation system includes the usual facilities for defining filled and justified text, sections, references and other report format operations; it interfaces not only to line printers but to more flexible printers including phototypesetters.

The PWB system is clearly a friendly environment in which to do conventional program and system construction. The power of the command language and the UNIX shell allow programmers a great deal of flexibility in organizing the construction process. While there are facilities that are difficult to provide with this architecture, (the authors specifically mention the absence of interactive debugging), there is no inherent limitation on the features that can be included in this system. Most of the tools provided are available in other good system construction environments; the integrated nature of this system makes the whole greater than the sum of the parts.

There is no central structure in PWB which records and controls system construction, and

no uniform method for describing systems. The programmer must, as in standard systems, provide the commands to build an externally represented system. (A recent addition to the PWB called Make provides some of these capabilities. Unfortunately, a complete description was not available in time to review.)

2.2.12 CLU

The CLU system [Lisk74b, Lisk74a] incorporates a facility for interconnecting modules as defined within CLU. Modules are either clusters or procedures, and each module has associated with it a *description unit* (DU) that contains all the computer processable information about a module other than the program for the module. Among that information are path definitions that are used for resolving module usage by a given module.

Description units include compiler information, such as the type identifier of the module (the name of the type in CLU that it implements), the types of parameters to the module, the module name, and the object code for the module. This amount of information makes possible strong type checking between modules when they are linked together. Descriptions units also contain documentation, specifications, debugging information, and a list of users permits notification of module changes.

The CLU system considers each module to be a part of its domain, and therefore has in fact a central data base of system information, including the modules and their DU's. The entire interconnection scheme of a system can be determined by tracing the system from the root DU, determining the set of modules used by it, and proceeding recursively on each of those modules.

The CLU mechanism permits the definition of multiple versions of modules in the sense that module definitions can be copied and the associated DU modified to include a different resolution of the modules used by that module. A slight reconfiguration of the scheme would permit multiple DU's for modules, increasing the flexibility of the mechanism. Systems could then be constructed by specifying the appropriate DU for each subsystem.

The definition of a module in CLU, however, restricts the interconnection mechanism to CLU-style modules. The definition of module used in this thesis is much broader than that of CLU, so the notion of module interconnection is also less restricted. Including the interconnection mechanism within a particular language will be seen to be an incorrect approach for the general problem of specifying system information interconnections.

2.2.13 Mesa

The Mesa system, developed at Xerox PARC [Gesc77], is primarily a programming language supporting a data type definition facility. Some attention, however, was paid to system construction as it affected the data type definition aspects of the language and considerable success has been achieved in providing type-safe intermodule linkage.

Modules, in Mesa, are either "definition" modules or "program" modules; some program modules "implement" definition modules. Programs that need resources provided by a module specify a definition module; later, a binding process provides an implementor for that definition. Type checking is performed as a part of binding. The authors of Mesa correctly emphasize the importance of separate construction tasks, such as compilation and binding. They solve the problem of plugging in multiple implementations of a given set of specifications, and for checking that the implementation provided "matches" the module that is expected by the caller.

The decision to address the construction issues from within the context of a particular programming language violates one of the principles of this research. Not only is it inevitable that other languages will be used (in non-programming contexts, if not other programming languages), but some information that is used in a system may not be naturally recorded in a programming language.

The Mesa mechanism also provides no facility for providing versions of system that vary in their specifications. Since providing multiple implementations of a single set of specifications is only a small percentage of the family variations, solving that problem results in only minor progress on the whole issue.

Finally, there is no design level description of the interconnections. Binding in Mesa is a dynamic searching process that follows "paths" that are established by the user who issues the "bind" command. The origin of system components may not in fact be easily determined once the system has been built because the path need not be retained. Furthermore, while module interfaces have been checked carefully, no assurance is available that global system design constraints have been obeyed.

(A librarian facility for Mesa is currently under development at Xerox. Some of the problems mentioned above will be addressed by that system).

2.2.14 Transformation Implementation

The Transformation Implementation (TI) system developed by Balzer[Balz76] is included here because the concept of verification it implements is similar to the notion of verification via construction. In that system, a programmer writes a rigorous, procedural, but abstract (with regard to representation of computation objects) program that is verified to perform the desired functions. Then, with assistance from the system, the user optimizes the program using source-to-source transformations that have been verified to preserve certain properties.

The basic approach, then, is to verify the original program and each of the transformations, and deduce from that collection the properties of the final, concrete program. Modifications to the programs are made to the original, abstract programs and the transformations, if still valid, are reprocessed. Later in this thesis, we will propose that various system properties be established not by examining the actual system components and "verifying" that they are correct, but rather by examining the information that was used to construct the systems and the mechanisms by which the construction was carried out, and from that deduce that the systems will exhibit the desired properties.

In general, it will be necessary to extend the transformational implementation approach to handle information that is originally represented in formats other that program text. Recording and reusing the transformations takes on great significance in the context of a system construction driver, especially the application of transformation rules to families of input information.

2.2.15 PLISS

White and Anderson describe a system for supporting construction of systems implemented in PL/I[Whit77]. The system controls the compilation of PL/I programs and maintains a data base about the interconnections between program segments. The system depends on use of a PL/I extension that permits the flexible definition of PL/I modules.

The authors consider the PL/I language insufficient for system construction due to the rigidity of the INCLUDE feature (which copies files into programs) and the lack of type checking on parameters of external functions. Therefore, the authors propose meta-modules that define PL/I modules in such a way that the system can check usage of external procedures. (A similar technique will be proposed in this thesis). In addition, a relational data base is used to store information about the interconnection of program segments. The actual programs and linkage files are stored in other files.

The central system construction data base is similar to that proposed in this thesis, although it is much too general and contains unnecessary and unduly constraining information. Unfortunately, many types of objects intuitively handled by programmers are not manifest in the data base. Having the system take partial control of construction is also compatible with the concepts of this thesis, as is the establishment of a "meta-language" to augment the virtues or overcome the limitations of the available language processors. The use of a graphical representation of system structure in this system is also appropriate.

The primary difficulty with using this system as a base is the myopic view of system construction adopted by the authors. The entire system is oriented toward the compile/link paradigm as available on IBM OS/360 systems (or MVS/370 systems) with PL/I as the programming language. Operating from within this world view prevents one from perceiving the more general information interconnections in systems, and the multitude of ways in that the information might be transformed other than compiling.

2.2.16 Software Engineering Data Base

Researchers at Softech have proposed a software engineering facility (SEF)[Irvi77]. An SEF is defined as an integrated collection of subsystems that assist the software engineering process, from requirements analysis to maintenance and enhancement of a system. The SEF as discussed in the reference is a proposal, not an existing system, although there is indication that some aspects of the SEF are implemented.

The central feature of the proposed SEF is a software engineering data base that contains all relevant information about the system in a machine processable format. It is intended that the various phases of development be sufficiently integrated into the data base that there is no need to "enter" the information into the data base from some external source, but that it naturally appears in the data base as the processed occur. In other words, the data base itself becomes the filing system, design document, record keeping system, and so forth.

Several subsystems process the data base in a variety of ways. A "requirements processor" facilitates the rigorous definition of system requirements. A general "system analyzer" is a facility for examining program structures, extracting information from the data base about system structures (e.g. total space requirements, dependencies among subsystems) and simulating control flow. An "interface auditor" checks module boundaries, verifies that subsystem boundaries are not violated, determines the effect (in terms of modified modules) of a modification, and records module histories. There are additional subsystems proposed for generating reports about the system and for applying test protocols. It is the view of the designers of the SEF that other subsystems can be integrated into this data base to extend the functionality of the SEF as a whole.

The work in this thesis contributes directly to the goals of an SEF. The central data base for software engineering is reintroduced in Chapter 5. It is appropriate to define the system information as a structured data base with a wide variety of processors to actually handle

the information, because the information is viewed very differently by different people in different phases of the development cycle. Furthermore, it will not be possible to predict the uses to which the data will be put, so that new subsystems will continually evolve to add new capability to the SEF.

It is also important to treat all documents of the software engineering task as flexibly and automatically as program text. The apparent immutability of documents normally printed on paper and read by humans compared to the relative plasticity of those read by machines reflects the effort that has been applied to the two domains. Recent work in programming languages has made the flexibility of programs even more remarkable; requirements documents, program logic manuals, user guides, system measurements, online help facilities, and other objects of relevance to the system, should also be integrated into the automated facility.

This thesis extends the SEF philosophy by "filling out" some of the details of the data base with some new concepts in program structuring. The SEF proposal designates that "system structure" be recorded in the data base, but no actual technique for representing "system structure" is presented. If a system structuring technique can be inferred from the description, it is a primitive mechanism with little abstract coherence. The necessity, within this scheme, for a subsystem that "checks" that subsystem boundaries have been honored, indicates that the subsystem definitions do not *control* the construction process. The system described in this thesis will have construction control completely dominated by subsystem description and such a tool would be superfluous.

The notion of sharing arises in the SEF proposal, but the authors do not analyze the nature of sharing. It is the opinion of this author that shared information is the most difficult problem in making feasible a system description, especially in the context of simultaneous multiple versions of information and systems.

2.2.17 Ad Hoc

Many of the specific goals of the work presented here have been achieved in particular cases by the use of ad hoc techniques. In fact, some of the motivation for this research has come from observing efforts of various clever technicians meet their needs with the tools available to them [Erma77, Rein77].

The following generalities are extracted from those observations:

- The methods that have resulted from those efforts have sometimes achieved fairly mundane goals with incommesurate amounts of ingenuity and computation. The success of the effort is likely to be appreciated less for its ostensible function than for the achievement of the function given the available facilities. A prototypical example is the introduction of type checking on parameters into a language that does not support such checking across compilation units, by using the compiler's macros facility to redefine procedure declaration syntax.

- Another characteristic of many of the ad hoc techniques is that their representation is convoluted, obtuse, fragile, and often not documented. These attributes derive not from the laziness of the programmer but from the awkwardness of the facilities.

- Application of similar techniques to new problems is difficult due to the lack of a uniform framework for representing the application of the techniques. A

programmer, attempting to decipher a piece of program or system text, must independently know the construction processes which will apply to it.

- Since the ad hoc facilities are embedded in other processes (e.g. macros processed during compilation), they are not easily separated from those other processes and are therefore hard to design and maintain. Without well established vocabularies of abstractions and transformations in the domain of system construction, the programmer promulgates those confusions. Considerable confusion attends the distinction between objects of the following pairs of types: files and directories, program text segments and facility or resource names, procedures and procedure names, "external" names and macro names, text editing commands and compilation instructions.

As mentioned in section 1.5, theory derives from practice. The greatest contribution to this work has been the (occasionally introspective) observation of programmers struggling to understand their activities and to implement tools that capture, within specific contexts, those intuitions. This thesis is primarily a generalization of the work of those people.

II System Descriptions

3. Software System Structuring Preliminaries

3.1 The Nature of a Software System

While there is much intuitive understanding about the nature of software systems and families of those systems, this chapter attempts to make explicit the model of software used in this thesis. It definitely does not include all software systems that currently exist, and may exclude a large class of useful software systems. However, it does accurately reflect the model of software that predominates in the "production" style of programming that characterizes almost all commercial, military, and computer vendor systems.

3.1.1 Construction from Components

One crucial characteristic of software systems is that they are constructed from original descriptive components, such as program segments. In other words, the systems can always be reconstructed from its source. These components are usually contained in files in the operating system environments, processed by compilers and other processors, producing the actual software system. The notation of the system components is usually completely independent from that of the resulting system (except possibly for debugging information that retains some of the original notation). The interface to the resulting system is supplied by that system, not some universal software. The executable portions of these systems are programs that run either directly on computer hardware (with or without assistance from an operating system) or are interpreted as if they were running on a special computer customized for the particular language.

Non-executable portions of systems are constructed in analogous ways, and while the resultant objects do not execute, they are stored in files, printed on paper, written onto a magnetic tape, transmitted across a communication link, or disposed of in some other (presumably useful) manner. For the duration of these sections, we will discuss systems as though they were comprised only of executable programs, although the reader is encouraged to remember that this is only to keep the prose from becoming ponderous with parallel alternatives.

3.1.1.1 The Nature of System Components. In conventional system construction environments, the components of a system are "source" files encoded in a particular programming language (or group of languages). There are usually two groups of source files: primary input to the compiler and secondary files that are included during compiler execution. Primary input files generally produce actual object code for a main program or group of procedures, or the definition of variables and constants. Secondary files define facilities; in some cases these are necessarily macro definitions, while in other cases they are macros (or similar definitional material) by convention. In some cases, the programmer specifies secondary files directly, while in others the compiler maps names to files.

The same view holds in state-of-the-art system construction environments, both real and hypothetical, although the secondary files have been replaced by a more rigorous definition facility that provides encapsulation of information. By exploiting the additional structure of the secondary files, the compiler and supporting programs can match specifications between module and user [Gesc77, Wulf76], permit multiple implementations of facilities [Gesc77, Lisk76], and preprocess auxiliary files [Wulf76, Gesc77, Lisk76].

The view of system component that is taken in this work is more general than either of the above. A system component is a file that contains some unit (or collection of units) of

information about the system. This information will probably be encoded in some way, but quite often not in a programming language, since that is an overly constrained and often unnatural representation of the information. For example, suppose the information contained in a system component is the definition of the finite state machine that characterizes the operation of some object in the domain of the program (e.g. identifiers in a language, illumination patterns of a traffic light, operation of an elevator). There are several standard representations for finite state machines, none of which is acceptable input to a programming language compiler. The conversion from one of those representations, such as an adjacency matrix, into a programming language might be a straightforward task, but clearly combines two pieces of information, namely, the particular finite state machine and its implementation in a particular language.*

There are many ramifications to generalizing the notion of system component. The "source" of a system is no longer presumed to be directly processable by a compiler; in fact, the compiler input files might well be generated by programs whose function is to combine various pieces of information into a syntactically valid program. The auxiliary files that contain definitional information are also encoded in ways natural to the information contained within them rather than in ways convenient to a compiler. A great deal of information in a system is concerned with the translation of information from one form to another.

3.1.1.2 Components vs Construction Information. As indicated by the preceding paragraph, there are two kinds of information in conventional systems. First, there are the components that contain algorithmic and data information. Second, there are the instructions by which these items are manipulated into a runnable (readable, distributable) system. The former are almost always encoded in files; the latter are seldom encoded, but are stored instead in the heads of the implementors or written on paper as instructions for human beings to execute.

Implementors increasingly record construction information in command files or command procedures that can be directly utilized by the operating system command language or the individual processors that perform the construction tasks. Rudimentary, however, is a mild word to apply to these facilities in the context of the information that they repute to represent. We can get an idea of the amount of effort that has been applied to the construction problems relative to the algorithmic problems of a system by contrasting the syntax, flexibility and appropriateness of even crude programming languages with that of the control languages for operating systems, linkage editors, text manipulations programs, compilers and other tools.

If encoding of system construction information is rare, however, encoding of design information is virtually non-existent. Design documents are almost always intended for human processing exclusively and the enforcement of design decisions is a tedious part of a project manager's job. There are a number of proposals for making system design descriptions more rigorous (for example, see [Ross77, Teic77]), but these are primarily intended for improved human-to-human communication.

Finally, given the rarity of machine processable representations of design and construction information separately, it should come as no surprise that there are no systems that provide for the integrated representation of design and construction information. In this work, we attempt to unify all three kinds of information within a single framework, so that design decisions can be exploited during the automated construction of systems built from

---

*Other cases are not so simple. Consider the amount of effort that has been devoted to converting decision tables into COBOL programs or machine code, or BNF grammars to parsers.

components that contain domain-specific information.

3.1.2 Manifestations of a System

In ordinary conversation about software systems, it is usually assumed that the manifestation of a system is a collection of executable programs and possibly some auxiliary data that is shared by some of the programs. In reality, however, the system takes many forms that have widely varying construction mechanisms. It is also clear that the concepts presented in this thesis, while derived from experience with software systems, may be applied to other objects constructed with similar processes, e.g. a complex document.

A system takes a very different form when it is distributed for installation at other computing sites. Depending on management policies and technical necessity, some collection of source objects, partially processed objects, documents, system construction procedures, instructions and test procedures will be packaged on a magnetic tape (for example). The tape certainly contains the information of the executable version, for it is intended that the executable version be created from it, but the process of packaging is different from that of linkage editing.

The information of a system takes yet another form in the documents for a system. Some of the information of a command language, for example, is reproduced in the user's manual for the system. The module designs, module interconnections, and various aspects of the components of the system are reproduced in the program logic manual. The operator's guide contains yet another collection of information. With current technology, documents are prepared by procedures very similar to software systems, and therefore we can clearly include documents in our scope.

In conventional environments, of course, the distribution and documentation of systems occur as activities independent of the software construction process. Where great expertise is devoted to the construction of the programs, the task of constructing the manuals is left to the people who can write but who may not know much about using the computer as a tool at all, much less exploiting the representation of the system components in the manuals. The distribution problem is generally handled ad hoc, with the massive exception of the SYSGEN [IBM72] and derivative mechanisms.

3.1.3 Information Sharing

As alluded to above, the construction of a system will involve the manipulation of information into forms suitable for compilation, document preparation, or whatever the immediate goal dictates. It could be the case that each piece of information was used in a particular manner in the system, such that we could consider the information units to have associated with them single transformations that were performed during the integration of the system. However, it turns out that information is often used in more than one way even in a single version of the system. For example, consider the information that associates with an aircraft type its seating arrangement. In one context, these arrangements may be bound to identifiers such as "SeatingBoeing727" as compile-time constants. In another context, the two values might be placed in separate fields of a record. In a third, aircraft types might be listed by total seating capacity.

One of the major problems in maintaining a system is ensuring that the various forms of the same information in a system remain consistent. Usually it is the responsibility of the programmer to systematically modify the various representations individually. In cases where the information is clearly contained in a single location, this task can be manageable, but even

then the programmer is charged with ensuring that this is the only place the information is defined.

3.2 The Nature of a Family of Software Systems

A family of software systems is a collection of systems that share design or implementation information. Families of systems evolve naturally. Although they are almost never planned, they arise from common situations. For example, the development version of a system coexists with the public version. Although the development version could be isolated from the public version, users often demand correction of errors and even additions of small features in the public version before the development version is ready for release. Hence, the two systems actually coexist in source form as well executable form. Maintaining consistency between these versions is difficult; customers of one large computer vendor complain that a bug found in release k (and for that a temporary fix is perhaps widely distributed) is fixed in release k+1 but reappears in release k+2 due to interleaved development phases.

System families evolve due to the parallel development of features by independent programmers, changes to perform measurements on the system, demands made by customers, and changes in the hardware or operating system. The designers of the implementation, however, tend to imagine the existence of only the "current" system and seldom make provisions for the multiplicity that actually occurs.

3.2.1 Dimensions of Variability

How are members of a system family related? The concept of system family is a very general and it is quite possible that a particular example of a technique could be misinterpreted to cover the entire notion. This has occurred often in the history of programming methodology: goto-less programming for structured programming, abstract data types for modules, synchronization primitives for the synchronization problem.

Replaceable subsystems are a significant and important technique for accomplishing system families for some types of variability. The replacement of one object module with another during linkage editing is a technique as old, relative to system construction history, as the closed subroutine in programming history. But providing replaceable subsystems, as has been quite flexibly and successfully done in [Gesc77, Lisk74a] does not result in the richness of system family desired here. In order to more fully condition the intuitions of the reader, there follows a partial list of dimensions along which two versions of a system might vary (possibly simultaneously). These dimensions are not completely independent, and considerable overlap exists among them.

- **Implementation Language.** Two versions of a system might be implemented in different programming languages. Consider a random number generator that has been programmed in Fortran, Pascal, PL/I, Snobol4, APL, Algol, Cobol and several other languages.

- **Underlying (Virtual) Machine.** Some systems are capable of exhibiting similar behavior on several computing systems. These systems might contain different computer hardware, operating systems, or versions of the operating system. The Snobol4 family of systems [Gris72] is a family of this form, as is the Pascal compiler [Wirt71]. This issue is generally given the stature of a separate problem called portability.

- **Tailoring.** When systems are sold to customers, they must be tailored to meet the particular needs and whims of the customer. Furthermore, these custom modifications must be maintained or enhanced through later releases of the system. Since customers seldom consider the conflicting needs of other customers, this form of system fragmentation is hard to control and often results in significant compromises with the design.

- **Instrumentation and Debugging.** If the behavior of a system is sufficiently complex or expensive, an effort may be undertaken to analyze the performance of the system. In some cases, program modifications are necessary to provide the data in a convenient form. These modifications must extend through the various sequential versions of the system but not exist in the production versions. A similar problem arises if additional program text must be added to provide debugging information, or if compilation and linking options must be specified to enable the built-in debugging functions of the language and/or operating system. These are, of course, independent of the instrumentation modifications and may need to coexist with them as well.

- **Flexible Specifications.** Overlapping or conflicting application environments for a system foster versions of the system with different specifications. The Sysgen [IBM72] associated with some operating systems allows the customer to modify the specifications by taking subsets of functionality, providing values for various system parameters, or naming alternative mechanisms for some system capabilities. A sysgen is more restrictive than tailoring since the options are specified from a menu of alternatives or attached to "hooks" at special points in the system.

- **Parallel Implementations.** Several implementations may coexist that meet a single set of specifications. Such implementations are often called "replaceable" or "repluggable" versions.

- **Configuration.** Even the same actual programs can be organized in different ways to provide different behaviors. For example, the distribution of data among the levels of the storage hierarchy could accommodate the system to various loads, constraints, or desired responsivenesses. Concurrent processes might be linked via shared storage, message queues, coroutine discipline, or a network. The functionality of the system does not change, but the methods for organizing the system are potentially quite different.

- **Substructure.** A simple version of a system might be appropriately implemented as a monolithic expanse of source program text. Without requiring any complex facilities, it may find the built-in resources of the programming language adequate for its task. The same problem with a larger scope might reasonably be subdivided, making use of library facilities such as symbol tables, file directories, indexed file organization, common command parsers and so on.

- **Experimental Systems.** Within a development or maintenance group, individuals might experiment with different portions of the system. Each of the different versions will be based on the "current" version of the system with a particular portion replaced by the new, improved information. Maintainers who are testing error repairs produce similar small deviation systems. These systems are characterized by their evanescense and, for some of them, their eventual

integration into other versions of the system.

- **Sequential Releases.** The most familiar family relationship among systems is succession. Every commercial system evolves through a sequence of "released" versions, possibly with intermediate update levels. This is the only dimension along which there is necessarily an ordering to the versions; for this reason it is particularly tempting to progress from one version to the next in a manner that destroys the previous versions, or makes them difficult to maintain. However, many pressures have developed in the field to make the simultaneous maintenance of several successive versions necessary. First, new releases are occasionally unstable and customers are unwilling to use them until they are seasoned. Second, many users modify their systems to accommodate their particular needs, and will only upgrade those modifications when a truly valuable new facility has been added or reliability has been significantly improved. Third, various systems in the installation may rely on idiosyncrasies of the early version that were deleted in later releases. Since users insist on error repairs and minor enhancements to "old" releases, systems must be actively maintained in several successive versions simultaneously.

- **Shared Subsystem.** Two systems can be related only because they share a subsystem.

Even though the dimensions along which versions vary may be reasonably independent, it does not follow that the implementations are independent. Therefore, the existence of three versions that vary along two dimensions does not imply that the "missing" fourth version could be implemented.

3.2.2 Documentation of Families of Systems

The documents for a single system also form a family. Consider a timesharing system with a quick reference card, a reference manual, an users' primer, a system maintenance manual, and a console operator's manual. The development and production of this family of documents is similar to the development and production of systems with parallel manifestations.

Generally the documents for a system exist in a single version, with the exception of successive documents for successive versions of the system. With the increasing plasticity of documents, it is possible to create a family of documents to reflect the family of systems. Suppose, for example, that there are several subsets of the command language for an operating system. The reference card for a user could be tailored to reflect the command language actually implemented at his site.

3.3 Information Sharing

3.3.1 Explicit versus Implicit Sharing

Some information in a system is shared implicitly among the components or versions of that system, i.e. by duplication, assumption, coercion or coincidence. Some information is explicitly shared; a tape record format encoded in Cobol and placed in a library is explicitly shared by the library pointer in the text of each user program. It is impossible to explicitly share all

information both because there is so much shared information in any system but also because each occurrence of explicit sharing is accomplished by an implicitly shared mechanism.

One of the difficult tasks of a designer is to determine what information to share explicitly and what to share implicitly. For example, suppose that the word size of the target computer is used in various places in the system. Should this value be tagged with an identifier, placed in a common file, and explicitly shared by each component of the system that makes use of it? If the program is a general purpose program that can operate on many different machines, this might be quite appropriate. On the other hand, if it is an operating system for a PDP-11, it is quite likely to be a nuisance that clutters up otherwise clear programs or, worse, an illusion that misleads the reader of a program into thinking that the program is independent of the word size.

This serves to remind us that the reason one wishes to control the use of shared information is that it may change or be redefined. Any information that cannot change is non-information (its information-theoretic content is no bits). The information for which explicit sharing is justified is that information that is likely to exist in more than one version, either simultaneously or successively.

The boundary between these two classes of information is occasionally fuzzy. Due to the difficulty of representing the information, or lack of rigor in utilizing it, explicitly shared information might also be implicitly shared. In addition, some information may be in the process of crossing the boundary; perhaps it has been implicitly shared but the process of collecting it together into a unit is not complete. Part of the process of redesign is moving pieces of information from one side of the boundary to another.

Information thought to be contained within a system sometimes becomes generalized or dispersed. For example, suppose a program initially stores some data in a format known only to itself. At some time, it becomes necessary to observe the operation of this program with an external monitor. The knowledge of how the data is stored may need to be made available to the monitor in order that it be able to measure the most relevant information. Similarly, when it becomes necessary to establish a parallel version of a system, almost all of the system information suddenly becomes shared between the two versions.

Such reorganization of information also results from the fact that systems with any large degree of novelty cannot be well designed before they are implemented. A designer may manipulate a set of ideas for several years before formulating the correct organization of those ideas. Obviously, the system will have been developed using the inferior design principles but should, if possible, be reorganized along the improved lines.

3.3.2 Common Representation

In order for information to be shared, it must be represented in a manner that is intelligible to all users of the information. One of the most important users of the information is the individual who is responsible for understanding and specifying that information. It would be nice if the information was intelligible to that user as well.

Programming languages are not always the most natural means for representing information. As those who have tried to do simple operations on program text can attest, even simple programming languages have grammars so complex that full lexical analysis and usually some syntactic analysis is necessary to perform such simple tasks as identifying procedure definitions, re-indenting or cross-referencing. (The presence of arbitrary macros is sufficient to prevent any significant action without complete text processing and parsing.)

Construction of program text, on the other hand, is quite straightforward.

In conclusion, shared information is best stored in a format that is natural for the human to interpret and modify, and that is easily processed to produce the specifically formatted representations necessary for specific purposes. It is likely that many of those transformations will be common across many applications; examples are the construction of records, case statements, arrays, parsers, tables, lists, and expressions. Other transformations will be for special purposes.

4. Software Family Description Concepts

4.1 Overview of Software Family Description Concepts

We will develop in this chapter a three-level software description notation that will bridge the gap between software design and software construction. The highest, most abstract, level of the notation defines the interconnections between subsystems or modules (there is no distinction). The intermediate level describes instantiations of system versions conforming to those interconnection structures. And the lowest, most concrete, level describes actual system construction operations built on a reasonable file and directory system.

We will propose a simple language with which we can describe many software families. However, we will not attempt to obtain a complete or even wholly consistent language with an elegant syntax and formal semantic specification; to attempt such a task without substantial experience in the use of these system description concepts is premature.

Subsystem Interconnection. The abstract portion of the notation corresponds to the subsystem interconnections languages proposed by DeRemer[DeRe76], Thomas[Thom76] and Tichy[Tich77]. Each *subsystem provides* a set of *resources* to other subsystems and *requires* a set of resources; in addition, is may be decomposed into constituent subsystems. Each subsystem obtains some of its required resources from the subsystems that constitute it (downward reference), others from *external* subsystems (horizontal reference) and the remainder from the subsystems of which it is a contituent (upward reference). Figure 4-1 shows a simple subsystem (PLOT) that provides a plotting facility using the trigonometric functions required horizontally from another subsystem (TRIG).

subsystem *PLOT* **provides** *PlotRoutines*
requires *TrigFcns* **external** *TRIG*
realization . . . end *PLOT*

Figure 4-1: PLOT--A Simple Subsystem

Although the division of subsystems leads naturally to tree or acyclic graph structures, it is possible to interconnect subsystems in arbitrary directed graphs. We presume that the system designer determines what subsystem relationships are reasonable and we do not in any way restrict the designer. As noted in section 7.1.2.3, cyclic use of subsystems does not necessarily result in either confusion or recursion.

Subsystem Versions. An actual, tangible, readable, runnable or demonstrable software system is an "instantiation" of some subsystem interconnection graph. We use the word version to denote such an instantiation, so two systems sharing an interconnection structure are defined to be versions of that subsystem. Because subsystems use resources from other subsystems, a specific version of one subsystem uses specific versions of those other subsystems. The instantiation level is the crux of the flexibility of this representation of software families rests; we shall devote considerable space to showing that it is adequate for describing many software system families. Figure 4-2 shows the previous example extended with a collection of versions. In this case, there is a version for each language in which the set of subroutines is implemented.

Subsystem Construction. The concrete level shows how the interconnection graphs and their associated versions are implemented using the underlying file system and the programs that

```
subsystem PLOT provides PlotRoutines requires TrigFcns external TRIG
    realization
        version Fortran . . . end Fortran
        version Algol . . . end Algol
        version Pascal . . . end Pascal
    end PLOT
```

Figure 4-2: PLOT--A Simple Version Collection Example

actually transform collections of characters into systems. Compilation, editing, linking and other construction processes are introduced as operators on *concrete objects*, some of which are source files. The abstract *resources* defined in the interconnection level are here reduced to mundane character strings; compilers and other programs integrate these character strings, creating the "core image," "document" or other files that are the final form of each system. (Due to forward referencing problems, this level will be discussed in detail in section 4.3, while section 4.4 will discuss system instantiation.)

A mechanism (*acquire*) will be introduced as a part of the construction level to exploit the flexibility of the interconnection and version descriptions. When a resource is needed during a specific construction process, the acquire function retrieves the proper string representation of that resource for the context from which it was requested.

In Figure 4-3 we have extended the previous example to include the construction information. Suppose that the plotting programs are used in source form in Algol, Fortran and Pascal programs. Let us assume also that there are programs that translate restricted Fortran to Algol or Pascal (including references to resources), and that <Plot Routine Source> is a file containing a Fortran program. FortranResolve is a program that inserts resources into Fortran program text. The file <Plot Routine Source> is outlined in Figure 4-4. The comment line beginning with "C#" is an instruction to FortranResolve and the translation programs to *acquire* the TrigFcns resource.

```
subsystem PLOT provides PlotRoutines requires TrigFcns external TRIG
    realization
        concrete object Original=file(<Plot Routine Source>)
        version Fortran select TRIG=Fortran.Source
            resources FortranResolve(Original) end Fortran
        version Algol select TRIG=Algol.Source
            resources FortranToAlgol(Original) end Algol
        version Pascal select TRIG=Pascal.Source
            resources FortranToPascal(Original) end Pascal
    end PLOT
```

Figure 4-3: PLOT--A Simple Concrete Construction Example

```
C     Plot Routine Source File -- Restricted Fortran with Resource Requirements
C
      Subroutine Plot1(A,B,C)
      . . .
      End
C
      Subroutine Plot2(D,E,F)
      . . .
      End
C
C#    Acquire(TrigFcns)
C
C     End of Plot Routine Source File
```

Figure 4-4: PLOT--A Simple Source File Example

4.2 The Interconnection Level: Subsystems

The *subsystem* is the basic building block of the description notation. In this section we will describe the structure of subsystems, define a syntax for defining them, and show how they can be interconnected. The following is the essence of the concept of *subsystem*.

> **A subsystem *provides* a set of *resources* that are available to other subsystems and may *require* a set of resources from other subsystems. An *internal* subsystem is one that is nested within another subsystem and provides its resources only to that enclosing subsystem.**

Resources are the currency of exchange among subsystems. They define how the information contained in a subsystem may be used outside that subsystem. A simple example of a resource is the interface description of a abstract data type. Resource formats can be designed to incorporate whatever information the designer deems appropriate.

As an example of a pair of interconnected subsystems, consider a subsystem that provides a pre-order tree traversal algorithm in the programming language Yfpl.* This resource might contain such operations as "traverse tree and apply function F to each node" and "traverse tree and collect balance statistics" encoded in a manner that is independent of a particular implementation of trees. Therefore, this subsystem *requires* a resource that defines a tree representation; that resource is provided by a subsystem that encapsulates trees.

Subsystems may contain other subsystems. The enclosing subsystem defines what aspects of the internal subsystems are provided by it to its users and establishes a common resource pool for the internal subsystems.

Each subsystem is instantiated as a set of *versions* (we will discuss these in detail in section 4.4). Therefore, each subsystem may represent several related "systems" as seen by users. We anticipate, for example, that there will be one STACK subsystem with several versions rather than several subsystems each providing a slightly different stack resource.

4.2.1 Specification of Subsystem Interconnection

The first part of a subsystem description defines the interconnection structure; the second

---

*Yfpl is, of course, Your Favorite Procedural Language, such as Algol, PL/1, Sail, assembler language or Fortran.

part, following the bracketing keyword *realization*, contains the definitions of the versions and the instructions for building them. This section addresses only the interconnection portion, for which the syntax is given in Figure 4-5.

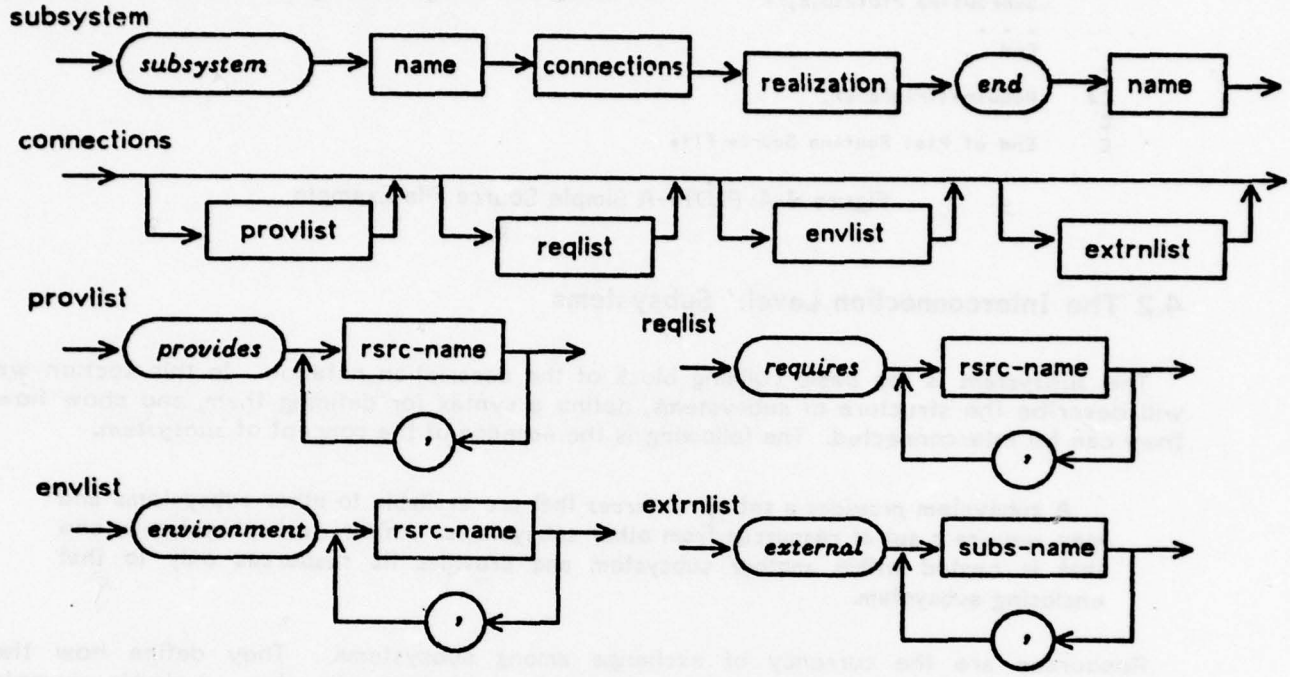


Figure 4-5: Subsystem Interconnection Syntax

Each subsystem has a name and three (possibly overlapping) sets of resources. The members of the first set are provided by this subsystem to its users. Those in the second set are required by this subsystem from other subsystems. The third set is an environment of resources available to all internal subsystems.

Each subsystem A also identifies those subsystems that provide the resources that A requires. Either those subsystems are nested within A (if they represent internal subsystems) or they are named in the *external* clause of A. In either case, the origin of any one resource is determined by the overlap of the resource lists associated with the juxtaposed subsystems.

4.2.1.1 Subsystem Interconnection Examples. For a simple example, consider a subsystem that provides sets of list operations but requires no other resources, as shown in Figure 4-6. We will refer often to this subsystem in later examples.

A more complex example is the database transaction and query system in Figure 4-7 that has been divided into user interface, transaction and output subsystems. The database subsystem provides a command language implementation and requires the resources

```
subsystem LIST provides ListBasic, ListSearch, ListApply
    realization . . . end LIST
```

Figure 4-6: LIST--A Subsystem that Provides Resources

implemented by the three internal subsystems.

```
subsystem DB provides CommandLanguage
    requires LanguageDefinition, Transaction, Query, Report
    subsystem CL provides LanguageDefinition realization . . . end CL
    subsystem XA provides Transaction realization . . . end XA
    subsystem RP provides Report, Query realization . . . end RP
    realization . . . end DB
```

Figure 4-7: DB--A subsystem with Internal Subsystems

The interconnections between the subsystems in the previous example result from the nesting of XA, CL and RP within DB, and the overlap between the requires list of DB and the provides lists of each of the nested subsystems. If a subsystem requires a resource that is not provided by one of its own internal subsystems, it must name the subsystem in its external clause. For example, the subsystem in Figure 4-8 uses resources provided by the independent subsystem LIST given above.

```
subsystem LISTUSER requires ListBasic, ListApply external LIST
    realization . . . end LISTUSER
```

Figure 4-8: LISTUSER--A Subsystem with an External Clause

If a subsystem S is organized so that all its internal subsystems should share resources common to the entire subsystem, then S may define a resource *environment*. Within S, a subsystem that requires a resource in the environment of S but does not specify whence it is provided, obtains that resource from S. In Figure 4-9, the list processing and symbol table resources are used everywhere within the subsystem S. The list processing resources come from the subsystem defined in Figure 4-6 whereas the symbol table resource comes from another internal subsystem of S.

```
subsystem S provides Useful1, Useful2
    requires ListBasic, SymbolTable, Other1, Other2
    environment ListBasic, SymbolTable external LIST
    subsystem ST provides SymbolTable realization . . . end ST
    subsystem S1 provides Other1 requires ListBasic, SymbolTable
        realization . . . end S1
    subsystem S2 provides Other2 requires ListBasic, SymbolTable
        realization . . . end S2
    realization . . . end S
```

Figure 4-9: S--A Subsystem that Defines a Resource Environment

In review, subsystems provide resources for use by other subsystems. There are three basic methods for interconnecting subsystems such that provided resources are available where required: a) a provider can be nested directly within a requirer, b) the provider can be named in a external clause in the requirer, and c) the provider can be named by a subsystem that encloses the requiring subsystem. In other words, the pool of resources available to a given subsystem are those provided by its internal subsystems, those provided by the subsystems named in its external clause, and those in the environment clauses of all subsystems that enclose it.

4.2.2 The Interpretation of Subsystem Interconnection Constructs

4.2.2.1 Subsystem and Resource Names. A subsystem name is an arbitrary tag that distinguishes this subsystem from all others. For subsystems that are not nested in any other subsystem, the names must be distinct. The names of nested subsystems need only be distinct within the subsystem in which they are nested; they are referenced by prefixing the enclosing subsystem name.

Resource names are likewise arbitrary tags that must be distinct only if a conflict would otherwise result. For instance, subsystems A and B could communicate a resource X as could subsystems C and D. The two X resources are independent.

4.2.2.2 Resource List Overlap. The set of resources provided by a subsystem may include resources that are also required it; in fact, resources marked with an asterisk in the provides list are automatically appended to the requires list. Resources that are both provided and required by a subsystem are considered to be the same resource passed transparently through that subsystem. In some cases, the user is unaware of the intermediary while in other cases the user can exploit it to gain flexibility. In the subsystem of Figure 4-10, resource α is provided to users by subsystem *A* but it is in fact provided by *AAA*.

```
subsystem A provides α*
    subsystem AA provides α*
        subsystem AAA provides α
        . . . end AAA . . . end AA . . . end A
```

Figure 4-10: A, AA, AAA--Transparent Resource Transmission

The environment list must be a subset of the requires list. This restriction simplifies the definition of acquire (see section 4.4.4) without removing any capability; a new internal subsystem can always be inserted to provide the necessary resources to the environment.

4.2.2.3 Subsystem Nesting and Scope of Names. Nesting is the textual indication that one subsystem is a part of another. Scope rules for internal subsystems are analogous to scope rules in languages with type definitions. Suppose that subsystem A' is textually nested within subsystem A, but that subsystem X is not, as shown in Figure 4-11.

- resources provided by A' are available to A.

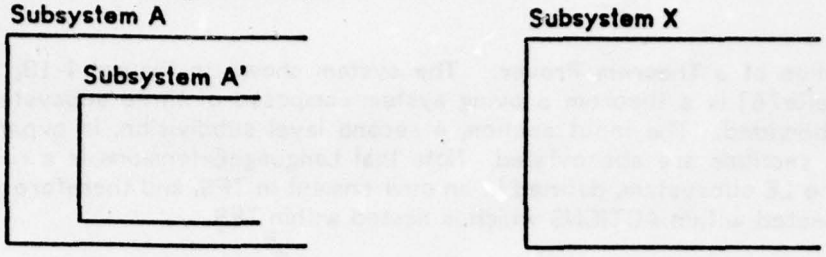


Figure 4-11: Nested Subsystems

- resources in an environment defined by A are available to A' but not to X.
- resources provided by A' but not by A are not available to X.
- resources provided by A' and both required and provided by A "pass through" and are available to X.
- If X must refer to A', it uses name A.A'.

The external clause of a subsystem A contains names of other subsystems in the block structured environment of A. Library subsystems, those available to all subsystems, are assumed to be defined in the implicit outer block. Subsystems nested within other subsystems must be referenced with a qualified name.

4.2.3 More Subsystem Examples

4.2.3.1 Symbol Table. Figure 4-12 shows a symbol table subsystem that requires resources HashFunction and ListBasic. Both those resources are provided by subsystems (HASH and LIST) which are in the library. The definition of LIST was given in Figure 4-6. Because ST names HASH and LIST in its external clause, and the resources in ST's requires list are found in the provides lists of HASH and LIST, ST is able to use resources HashFunction and ListBasic.

subsystem *ST* **provides** *SymbolTable*
requires *HashFunction*, *ListBasic* **external** *HASH*, *LIST*
realization . . . end *ST*

subsystem *HASH* **provides** *HashFunction* **realization . . . end** *HASH*

Figure 4-12: ST--Symbol Table using Library Resources

4.2.3.2 Input Section of a Theorem Prover. The system shown in Figure 4-13, transcribed from DeRemer[DeRe76] is a theorem proving system composed of three subsystems each of which is also subdivided. The input section, a second level subdivision, is expanded in full below; the other sections are abbreviated. Note that LanguageExtensions is a resource that is provided by the LE subsystem, defined in an environment in TPS, and therefore available to INPUT which is nested within ACTIONS which is nested within TPS.

```
subsystem TPS provides . . .
    requires LanguageExtensions, . . . environment LanguageExtensions, . . .
    subsystem LE provides LanguageExtensions . . . realization . . . end LE
    subsystem CLAUSES . . . realization . . . end CLAUSES
    subsystem ACTIONS provides . . . requires . . .
        subsystem THM . . . realization . . . end THM
        subsystem OUTPUT . . . realization . . . end OUTPUT
        subsystem INPUT provides InputParser
            requires Scanner, Parser, PostProcessor, LanguageExtensions
            subsystem PARSE provides Parser
                requires Scanner, LanguageExtensions
                external SCAN
                realization . . . end PARSE
            subsystem SCAN provides Scanner
                requires LanguageExtensions
                realization . . . end SCAN
            subsystem POST provides PostProcessor
                requires LanguageExtensions
                realization . . . end POST

            realization . . . end INPUT
        realization . . . end ACTIONS
    realization . . . end TPS
```

Figure 4-13: TPS--Theorem Prover using a Resource Environment

4.2.3.3 KWIC Index System. The example in Figure 4-14, transcribed from Thomas[Thom76] who obtained it from Parnas[Parn72d], represents a KWIC index system that is subdivided into an subsystems for input, line storage, circular shifting, alphabetizing and output.

```
subsystem KWIC provides Kwic requires InputLine, Alph, Ith, OutputLine

    subsystem INPUT provides InputLine
        requires Line, InputErrorHandler external LS, IEH
        realization . . . end INPUT

    subsystem LS provides Line
        requires StorageErrorHandler external SEH
        realization . . . end LS

    subsystem ALPH provides Alph, Ith
        requires CsSetup, ShiftedLines external CS
        realization . . . end ALPH

    subsystem CS provides CsSetup, ShiftedLines
        requires Line external LS
        realization . . . end CS

    subsystem OUTPUT provides OutputLine
        requires ShiftedLines, Ith external Alph, CS
        realization . . . end OUTPUT

    subsystem IEH provides InputErrorHandler realization . . . end IEH
    subsystem SEH provides StorageErrorHandler realization . . . end SEH
    realization . . . end KWIC
```

Figure 4-14: KWIC--Resource Usage among Internal Subsystems

4.3 The Construction Level: Concrete Objects

A system implementor must specify, directly or indirectly, the processes by which a system is constructed. In this section, we provide a notation for specifying common varieties of these processes. Figure 4-15 shows the syntax for construction processes and the following are definitions of the terms employed in this section.

A *concrete object* is a generalized file. Some concrete objects are in fact the source files of a system. Others are the intermediate objects such as the output of a compiler or macro processor. Still others are the final system objects such as executable machine code or a formatted document.

A *file* is a member of the file system on which interpreters of this notation operate. We use them only as repositories for source programs and data portions of other concrete objects.

A *processor* is any program that produces a concrete object. Usually it has at least one parameter that is also a concrete object. Common processors include compilers, assemblers, cross reference generators, linkage editors, document formatters, sort programs, printing programs, macro expanders and text processors.

A *rule* shows how a concrete object is constructed. Rules are functional in form and may be nested. Special rules are used to coerce resources and files into concrete objects.

A given concrete object may be the result of several construction processes, and therefore is a maximal element in a lattice of concrete objects. The minimal elements of this lattice are called *source files.*

As we will see in section 4.4, concrete objects will belong to (versions of) subsystems. Although the construction lattice of a given concrete object can include concrete objects from many different subsystems, access to concrete objects is carefully controlled.

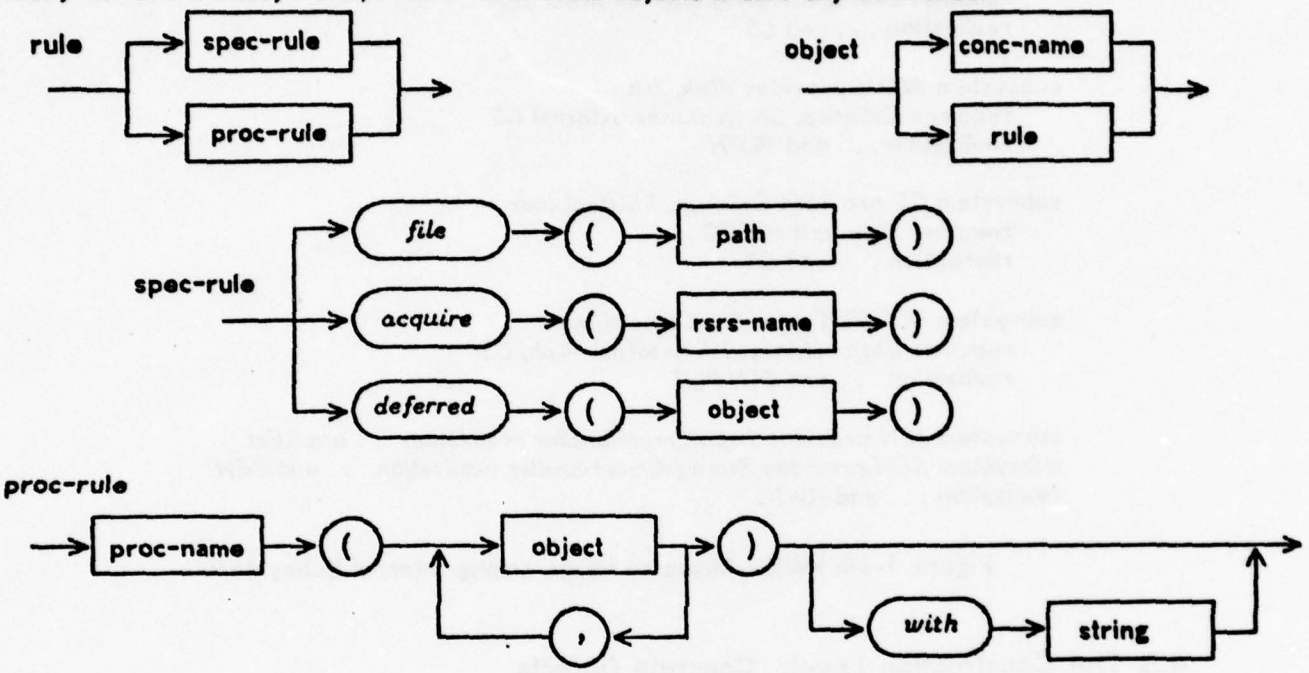


Figure 4-15: Construction Process Syntax

4.3.1 Rules for Primitive Concrete Objects

The parameter to the *file* operator is a file name. We will use a name within "<>" brackets to represent a full directory path in the file system supported by the operating system; this may include a multi-level name, a project identifier, a library identifier, and/or an account number. It is important here only that it uniquely specify a single file. For very short files, we will occasionally write in quotation marks the contents of the file in place of the file specification. This is interpreted to mean the name of a file containing the given character string.

file(<*Source File Name*>)
"*just a few characters*"

The *acquire* operator converts a resource from another subsystem into a concrete object

in the local context. Although the *resource* is a primitive concrete object in this local context, it may have been the result of a complex construction process in its own subsystem.

```
acquire(UsefulResource)
```

4.3.2 Rules for Compound Concrete Objects

New concrete objects are created by processing others. If a program is written in language Yfpl, the Yfpl compiler would produce a new concrete object from the file containing the program source. Some processors have several concrete object parameters.

```
Yfpl(file(<Program Source>))
Yfpl( "begin print( 'done') end ")
Merge(file(<Original Source>), file(<Updates>))
```

These rules can be embedded as deeply as desired. Suppose there are preprocessors which convert management specifications to decision tables, decisions tables to branching networks, branching networks to block structured Fortran, block structured Fortran to ANSI Fortran, and ANSI Fortran to PL/I. The rule for processing a management specification into a runnable program would be as follows:

```
Link(Pl1(AnsiToPl1(BsfToAnsi(BnToBsf(DtToBn(MsToDt(file(<Manage Specs>)))))))
```

It is often convenient to provide additional information to a processor with a set of strings. Suppose, for example, that if a program is included in the debugging version of a system, the compiler must be instructed to include symbols and debugging linkage. The ordinary and debug rules follow.

```
Yfpl(file(<Program Source Text>))
Yfpl(file(<Program Source Text>)) with "Debug"
```

4.3.3 Deferred Concrete Objects

Although we have formulated construction as a completely explicit sequence of functions, some concrete objects participate implicitly in construction processes. Separately compiled subroutine bodies are the prototypical example. During compilation of a program that uses a collection of subroutines, the compiler will use a resource containing external procedure declarations. The use of that resource implies that later, during linkage editing, the compiled code bodies of the procedures will be needed. The *deferred* operator retrieves all objects that have been implicitly associated with its parameter object.

In the context discussed in the next section, we can give names to concrete objects; the following represents the construction of a program which uses a resource that has deferred objects associated with it:

```
concrete object UserProgram=YFPL(file(<User Program Source>))
concrete object UserExecutable=Link(UserProgram,deferred(UserProgram))
```

4.4 The Instantiation of Systems: Versions

Real systems are tangible objects; they may be moved, read, executed, or in some way manipulated. The interconnection structure of a subsystem does not show this corporeal aspect of real systems, while a collection of concrete objects, in isolation, does not capture

the *gestalt* of a system. We are in the gap between the blueprints and the bricks, in need of a way to describe contracting. We must bridge the chasm between information relationships and bits reflecting the intuitions we have about how software technicians view the objects with which they work.

> **The *realization section* contains all the information pertinent to the tangible form of a subsystem, primarily a list of *versions*.**
>
> **A *version* is an instantiation of a subsystem or a group of such instantiations. Each subsystem has at least one version. Each version consists of several collections of concrete objects that are its physical form.**

Because system versions share the same interconnection structure, we prevent duplication of identical information, clearly portray differences between versions, and centralize modification sites. If we were instead to copy system descriptions and perform small alterations, modification of the shared portion would necessitate modifications to each copy of that portion.

There are three visible collections of concrete objects in a version: a) objects essential to the existence of that version of the subsystem, b) objects that represent resources, and c) objects whose use is associated with the use of a resource (deferred objects). For example, consider a subsystem that provides a central error message and diagnostic facility for a system. The essential object for a version of the subsystem is the file containing the error message texts; the resource is represented by a file containing external procedure declarations; and the actual code bodies for those procedures are an object whose use is associated with the use of the resource.

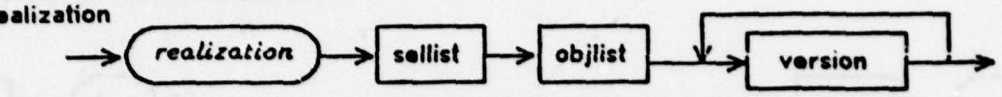
4.4.1 The Specification of System Versions

4.4.1.1 Individual Versions. The instantiations of a subsystem are contained in the second section of the subsystem notation following bracketing keyword realization (see Figure 4-5). The syntax for the realization section is given in Figure 4-16. The primary component of the realization section is the list of versions of the system; the syntax for version definitions is given by Figure 4-17. Each version has a name, unique within the subsystem, and a collection of information that distinguishes this version from all other versions of the subsystem. In addition, versions can be organized hierarchically, although we will postpone an example of this until section 4.4.1.5. Figure 4-18 shows an example subsystem with three versions.

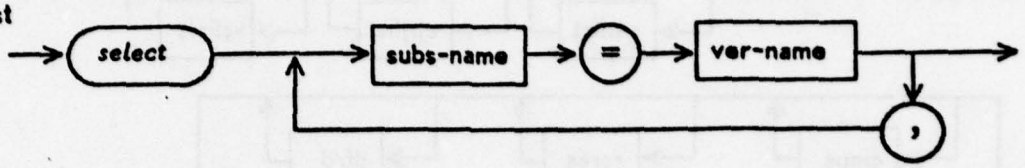
```
subsystem COMPILER provides . . . requires . . .
    realization
        version Backup . . . end Backup
        version Current . . . end Current
        version Development . . . end Development
    end COMPILER
```

Figure 4-18: COMPILER--Subsystem with Three Versions

realization



sellist



objlist

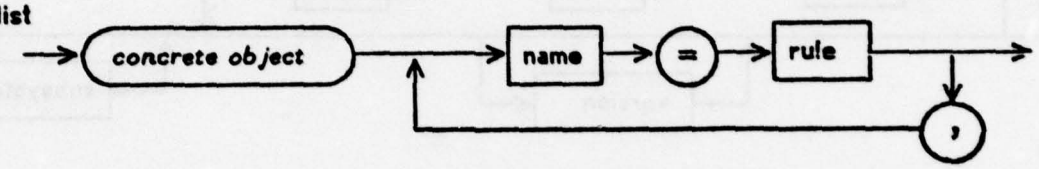


Figure 4-16: Realization Section Syntax

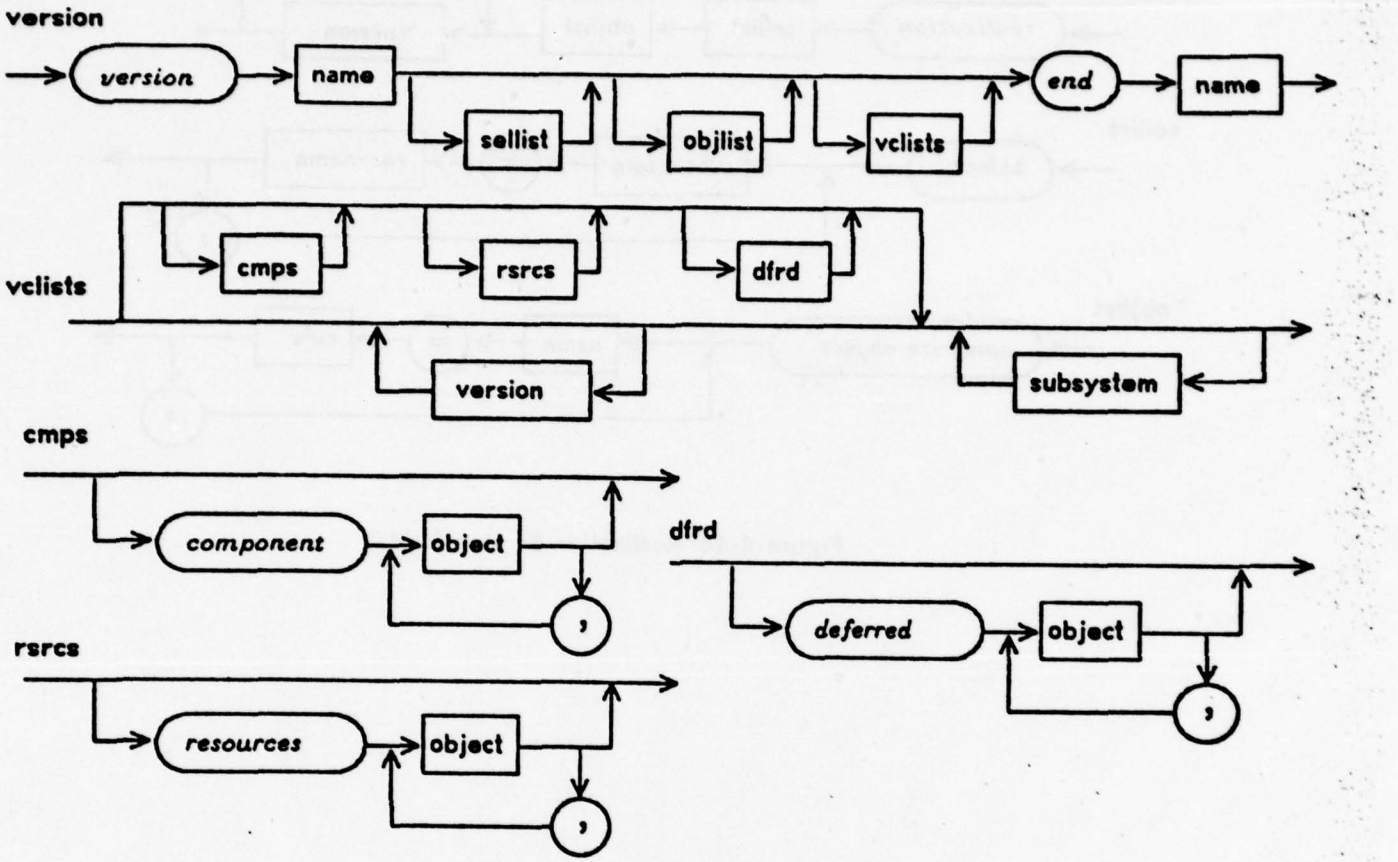


Figure 4-17: Version Syntax

4.4.1.2 Selection of Version. During construction within a version, resources are obtained from other subsystems, each of which may have multiple versions. The select clause specifies a version for each relevant internal subsystem and external subsystem. It may appear within the realization section, in which case it applies to all versions, or within a version, in which case it applies only to that version and those hierarchically below it. The example of Figure 4-19 shows simple version selection of both forms. In this case, version A1 of subsystem A and version B2 of subsystem B will be used by both versions of MAIN1. However, version C3 of subsystem C will be used in the first version, whereas version C4 will be used in the second version. .

Selection of version for subsystems that supply environment resources must be made in the realization section of the subsystem that establishes the environment, since the name of the supplying subsystem is not known in those subsystems in which the resources are actually used. In Figure 4-20, subsystem ENV provides the environmental resources and its

```
subsystem MAIN1 requires rsrca, rsrcb, rsrcc external A, B, C
    realization select A=A1, B=B2
        version first select C=C3 . . . end first
        version second select C=C4 . . . end second
    end MAIN1
```

Figure 4-19: MAIN1--Version Selection Example

version is selected at the MAIN2 realization level, whereas REG provides regular resources and the version selections can appear within the versions.

```
subsystem MAIN2 requires Env1, Env2, reg1, reg2
    environment Env1, Env2 external ENV, REG
    realization select ENV=Env
        version Main2a select REG=Reg1 . . . end Maina
        version Main2b select REG=Reg2 . . . end Mainb
    end MAIN2
```

Figure 4-20: MAIN2--Environments and Version Selection

4.4.1.3 Definition of Concrete Objects. Versions of subsystems include collections of concrete objects, as suggested in section 4.4. A concrete object may be defined either where it is used, or within the version to which it contributes, or within the realization section of its subsystem. Those objects accessed by the subsystem's users are divided into three categories, those that are representations of resources, those that are associated with the use of a resource, and those that are essential parts of the version.

When a resource is actually requested from a subsystem, a particular version of the subsystem will be specified. The *resources list* of concrete objects within that version specifies, for each resource, which concrete object contains its representation. For example, within version Abc1 in Figure 4-21, we see that resource a is represented by concrete object x1.

```
subsystem ABC provides a, b, c
    realization
        version Abc1 concrete object <definitions of x1, y1, z1>
            resources a=x1, b=y1, c=z1 end Abc1
        version Abc2 concrete object <definitions of x2, y2, z2>
            resources a=x2, b=y2, c=z2 end Abc2
    end ABC
```

Figure 4-21: ABC--Resource Representation Example

If a single concrete object represents a single resource, the associations between the two may be specified as above. In cases where a single concrete object represents multiple resources, they may be grouped in parentheses. If all resources are contained in a single concrete object, the resource names can be omitted. In the first example below, both a and b

are represented by x1; in the second example, all resources are represented by x.

```
resources (a, b)=x1, c=x2
resources x
```

Concrete objects that must be integrated into a system later, but are associated with the use of resources, are called *deferred* objects and are listed like resource objects, except that each resource may have several objects associated with it. Each object in the deferred list for a resource is attached to the requesting object as a deferred object when that resource is used. For example, if a resource is represented by a set of external procedure declarations, the compiled code for the procedures is attached to the using object until a linkage editing program can assimilate it. See section 4.4.4 for more details on this mechanism. In Figure 4-22, object w1 is associated with the use of any resource from version Abc1.

```
subsystem ABC provides a, b, c
    realization
        version Abc1 concrete object <definitions of x1, y1, z1, w1>
            resources a=x1, b=y1, c=z1 deferred (a, b, c)=w1 end Abc1
        version abc2 concrete object <definitions of x2, y2, z2, w2>
            resources a=x2, b=y2, c=z2 deferred (a, b, c)=w2 end abc2
    end ABC
```

Figure 4-22: ABC--Deferred Object Example

The *components* of a version of a subsystem are independent of the use of resources. Whenever a version of the system is built, each component will be constructed. In Figure 4-23, x4 and y4 are objects that are a part of version Abc2 but are not related to resource usage.

```
subsystem ABC provides a, b, c
    realization
        version Abc1 concrete object <definitions of x3, y3>
            components x3, y3 end Abc1
        version Abc2 concrete object <definitions of x4, y4>
            components x4, y4 end Abc2
    end ABC
```

Figure 4-23: ABC--Component Object Example

4.4.1.4 Substructure of a Version. One version may basically share the structure of other versions, but perhaps require additional resources. For example, the instrumentation version might require access to the resources provided by a data collection subsystem. In these cases, the additional subsystems can be included only within the relevant version. Any resource provided by such a subsystem is assumed to be required by the version. The select clause for the version must also include selections for these subsystems. In Figure 4-24, subsystem S1' is a conduit bringing the Collect resource from INST to the Instrumented version of S1 without impacting the requirements of the Regular version.

```
subsystem S1 requires Rsrc3 provides Rsrc1, Rsrc2 external S2
    realization
        version Regular select S2=xx . . . end Regular
        version Instrumented select S2=xx, S1'=Fortran
            subsystem S1' provides Collect* external INST
                realization version Fortran select INST=Fortran end Fortran
                . . . end S1'
            . . . end Instrumented
    . . . end S1
```

Figure 4-24: S1--Subsystems within Versions

4.4.1.5 Hierarchical Organization of Versions. Versions may be organized into trees. Only leaves of the tree can contain the resources, deferred and component lists; non-leaf versions may contain select clauses and concrete object definitions that apply to all versions in that subtree. All versions may contain subsystem definitions. A particular version is denoted by the path of nodes in the version hierarchy. For example, the quick Fortran version of the subsystem in Figure 4-25 would be designated "Quick.Fortran".

```
subsystem HASH provides HashFunction
    realization
        version Quick
            version Fortran resources file(<Fortran Quick Hash>) end Fortran
            version Pascal resources file(<Pascal Quick Hash>) end Pascal
            version Algol resources file(<Algol Quick Hash>) end Algol
            end Quick
        version Careful
            version Fortran resources file(<Fortran Careful Hash>) end Fortran
            version Pascal resources file(<Pascal Careful Hash>) end Pascal
            version Algol resources file(<Algol Careful Hash>) end Algol
            end Careful
    end HASH
```

Figure 4-25: HASH--Version Trees

4.4.2 The Interpretation of System Instantiation

4.4.2.1 Definition of Concrete Objects. A concrete object may be defined in the realization section of a subsystem or at any level in the version hierarchy. Concrete objects may use only those resources that are available at the same level, that is, resources from subsystems that have been selected at that or a higher level. This restriction is necessary because a concrete object that uses one definition of a resource usually differs from a concrete object that uses another definition.

There are cases, however, in which two definitions are equivalent and therefore result in identical concrete objects. For example, two versions of externally compiled procedures are

invoked from a program using identical external procedure declarations; the resource objects will be identical but the deferred objects will differ. We rely on an optimization-style analysis to allow us to avoid reconstruction of such identical objects.

4.4.2.2 Scope of Names and Selections. There are three ways in which the version of a subsystem can be specified. First, it may be named in the select clause of a specific version at any level. Second, it may be named in the realization section of the subsystem. Third, it may be named in the realization section of a subsystem in which this one is nested; this last option is used only for environment definitions. Usually only one specification will apply. If there is a conflict, the "closest" specification is used; version-specific selections override global selections, which in turn override nested selections.

Concrete objects are local to the subtree in which they are defined. If necessary, concrete objects defined in subsystems within which the current one is nested may also be used, provided they are not within a version in that subsystem.

4.4.3 Examples of System Instantiation

4.4.3.1 In-line vs Out-of-line Implementation. A set of functions may be compiled in-line, as with macros, in which case the code for the functions is inserted into the context of each call. Alternatively, they may be compiled out-of-line, in which case they would be invoked using the procedure call mechanism of the programming language. If the language does not accomodate this flexibility internally, different program text for the functions is needed for each of these cases. The construction of the concrete objects indicated by the text in "{}" in Figure 4-26, which illustrates this example, should follow a paradigm like that proposed in section 4.5.2.

```
subsystem FCN provides Functions
    realization concrete object FcnDef=file(<Function Source Text>)
        version InLine
            resource {construct macros from FcnDef}
            end InLine
        version OutOfLine
            resource {construct external declarations from FcnDef}
            deferred Yfpl({construct procedures from FcnDef})
            end OutOfLine
    end FCN
```

Figure 4-26: FCN--In-line/Out-of-line Procedures

4.4.3.2 Alternative Specifications. Suppose that a sort facility can be efficiently implemented without recourse to disk storage if the amount of data it must accomodate is limited, whereas in general disk storage would be required. The disk version requires additional library facilities provided by the RA subsystem. Figure 4-27 contains a description of this system.

4.4.4 Representation Exploitation Mechanism: Acquire

The crux of our scheme for representing systems is the flexible combination of resources

```
subsystem SORT provides SortFile
    realization
        version TinyFiles resources file(<In Core Sort>) end TinyFiles
        version HugeFiles select RA=inline
            resources {construct sort program}
            subsystem RA provides RandomAccessDisk
                realization
                    version InLine . . . end InLine
                    version OutOfLine . . . end OutOfLine
                    end RA
            end HugeFiles
    end SORT
```

Figure 4-27: SORT--Parallel Alternatives

at the concrete level, along lines established at the subsystem level, according to alternatives presented at the version level. The mechanism that exploits this flexibility is called "acquire" and is available as a processor for converting resources to concrete objects, and as a subroutine for to processors that run under the control of the system that monitors construction. As a processor, acquire obtains the representation of a resource and makes it available, for example, as input to another processor or another resource definition. (The use of acquire by processors will be discussed in Chapter 5.)

Suppose that an object that provides a resource from subsystem A is constructed by running program Munge on a resource provided to it by subsystem B, as shown in Figure 4-28. Here acquire is used as a processor to coerce y, a resource, into a concrete object as needed by Munge. The output of Munge is the representation of the Aonly version of the resource x that is provided by A.

```
subsystem B provides y
    realization
        version Bonly
            resources file(<definition of y>)
            end Bonly
        end B
subsystem A provides x requires y external B
    realization
        version Aonly select B=Bonly
            resource Munge(acquire(y))
            end Aonly
    end A
```

Figure 4-28: A,B--Acquire as a Processor

4.5 Complete Examples

The following sections present some examples that clarify the operation of acquire and the use of versions.

4.5.1 Name/Value Pairing

Device descriptions are a part of many operating system interfaces. If they are not available to a normal user, they are probably used between levels within the operating system. These descriptions may have many attributes, subdivided in abitrarily complex ways. For example, limiting the domain to disk-style devices, a programmer could wish to exploit such information as the number of tracks per cylinder, the hardware block size, the number of heads, the physical address of the device, the capacity of a pack, the paths by which it may be accessed, whether the device is protected against writing, and so forth. Similarly, confining our attention to terminals, the description would include page width, character set, direct cursor addressing capability, memory capacity, hardcopy capability, and so forth.

Without loss of generality, assume that there is a bit pattern representation of each device type within the group, and a corresponding name and code. For example, for a collection of terminals we could have the values given in Figure 4-29. For the time being, we shall assume that there is no decomposition of this bit pattern that is meaningful, only that there is a one-to-one correspondence between names and bit patterns.

Code	Name	Bit Pattern
M33	Model 33 Teletype	000000010001
M37	Model 37 Teletype	000000110001
IBM	IBM 2741	110000010001
INF	Infoton CRT	011000101000
BEE	Beehive CRT	001000101000
3RG	3 Rivers Graphics	111111111110
ADM	ADM 3 CRT	101000011000
TI7	TI Silent 700	000010010001
POD	Diablo	000100010001

Figure 4-29: Terminal Type Bit Patterns

For our first example, let us hypothesize that two programs use this information. The first needs a set of identifiers, one for each type of terminal, to test the terminal type (as returned by the operating system) and decide how much output to print. The other program prints the terminal type using a pair of vectors in which it looks up a bit pattern to find a corresponding string description. The general structure for this facility, omitting the actual construction rules, is given in Figure 4-30. The subsystems that use this facility are shown in Figure4-31. The file <Program1 Source> contains a program with the skeleton shown in Figure 4-32. The file <Program2 Source> contains a program with the skeleton shown in Figure 4-33.

In each program, the declaration "ACQUIRE TerminalTypes" instructs the compiler to replace that declaration with the representation of the resource "TerminalTypes". In the first case, the program expects that "TerminalTypes" will cause the declaration of identifiers of the form "Tt" followed by the conventional three character code for each terminal. The second program, however, expects that two vectors, "TtBitpattern" and "TtName", both of length available in the identifier "TtNum", be declared and initialized to have the obvious values in corresponding entries.

```
subsystem TT provides TerminalTypes
    realization
        version Identifiers resources TerminalTypes={rule} end Identifiers
        version Vectors resources TerminalTypes={rule} end Vectors
    end TT
```

Figure 4-30: TT Subsystem Configuration Skeleton

```
subsystem PROG1 requires TerminalTypes external TT
    realization
        version Prog1 select TT=Identifiers
            component Yfpl(file(<Program1 Source>))
            end Prog1
    end PROG1
subsystem PROG2 requires TerminalTypes external TT
    realization
        version Prog2 select TT=Vectors
            component YFPL(file(<Program2 Source>))
            end Prog2
    end PROG2
```

Figure 4-31: TT User Subsystems

```
 BEGIN "program 1"
DECL Terminal:integer, Style:(Terse, Normal, Verbose);
. . .
ACQUIRE TerminalTypes;
. . .
Terminal ← GetTerminalType();  Comment operating system call;
. . .
IF Terminal = TtM33 OR Terminal = TtM37 OR
   Terminal = TtPOD OR Terminal = TtTI7 OR Terminal = TtIBM
   THEN Style ← Terse;
IF Terminal = TtBEE OR Terminal = TtADM OR Terminal = TtINF
   THEN Style ← Normal;
IF Terminal = Tt3RG THEN Style ← Verbose;
. . .
END "program 1"
```

Figure 4-32: Contents of <Program1 Source>

```
 BEGIN "program 2"
DECL Terminal:integer, TerminalName:string;
. . .
ACQUIRE TerminalTypes;
. . .
Terminal ← GetTerminalType();  Comment operating system call;
. . .
TerminalName ← "unknown";
FOR I IN [1..TtNum] DO IF Terminal=TtBitpattern[I]
   THEN TerminalName ← TtName[I] OD
WriteLog("User Terminal Type: ", TerminalName);
. . .
END "program 2"
```

Figure 4-33: Contents of <Program2 Source>

How might these two versions of the resource be provided? The details of four possible methods will be defined below; each is an advance along information hiding lines over the one before. In the first case, each of the two versions of the resource is contained in a separate file. A slight improvement results from combining them into a single file, and using a program to extract the proper version. The next step is to store one version, and use a program to convert it to the other version. Finally, we store a canonical form of the resource, and use programs to generate each of the two actual resources.

Figure 4-34 show the complete TT subsystem for the first case, wherein the two versions of the resource are maintained in separate files. The contents of file <Terminal Type Idents> would be of the form shown in Figure 4-35. The CONST notation is derived from Pascal, and the "!" character begins a "here-to-end-of-line" comment. The result of including this segment into a program is to declare several identifiers and bind them to the integers represented with binary literals. The contents of file <Terminal Type Vectors> would contain this information in the form shown in Figure 4-36. Again, CONST indicates compile-time binding; the structures are assumed to be arrays due to the index type provided in the declaration, and the initial values of the array elements are provided sequentially following the "=" initialization specifier.

```
subsystem TT provides TerminalTypes
    realization
        version Identifiers resources file(<Terminal Type Idents>) end Identifiers
        version Vectors resources file(<Terminal Type Vectors>) end Vectors
    end TT
```

Figure 4-34: First TT Subsystem Configuration

```
CONST TtNum:integer=9;

CONST TtBitpattern[1..TtNum]:integer=(#00010001, #00110001, #11010001, #00101000,
      #00111000, #11111110, #10111000, #10010001, #00011111);

CONST TtNames[1..TtNum]:string=("Model 33 Teletype", "Model 37 Teletype", "IBM 2741",
      "Infoton CRT", "Beehive CRT", "3 Rivers Graphics", "ADM 3 CRT",
      "TI Silent 700", "Diablo");
```

Figure 4-36: Terminal Types Array Resource

```
CONST
   TtM33: integer=#00010001,        | "Model 33 Teletype"
   TtM37: integer=#00110001,        | "Model 37 Teletype"
   TtIBM: integer=#11010001,        | "IBM 2741"
   TtINF: integer=#00101000,        | "Infoton CRT"
   TtBEE: integer=#00111000,        | "Beehive CRT"
   Tt3RG: integer=#11111110,        | "3 Rivers Graphics"
   TtADM: integer=#10111000,        | "ADM 3 CRT"
   TtTI7: integer=#10010001,        | "TI Silent 700"
   TtPOD: integer=#00011111;        | "Diablo"
```

Figure 4-35: Terminal Type Identifier Resource

This method is primitive because both files must be kept up to date individually, and the addition of a terminal type to one does not guarantee that it will be added to the other. If we can encourage the programmer to see the two files as a single file, we can increase the probability that the programmer will change one when the other is changed. Suppose that the two files above are separate pages of a single file, and that the program Extract extracts individual pages of files. The revised subsystem for TT, shown in Figure 4-37, reflects this new concrete object configuration.

```
subsystem TT provides TerminalTypes
    realization
        concrete object TrmTypes=file(<Terminal Type Definitions>)
        version Identifiers resources Extract(TrmTypes) with 1 end Identifiers
        version Vectors resources Extract(TrmTypes) with 2 end Vectors
    end TT
```

Figure 4-37: Second TT Subsystem Configuration

Now the two versions are linked by reference to the common file. Even this textual adjacency cannot guarantee that the two pages will be correctly maintained in parallel. It would be reasonable to store only the first file, with the assignment statements, and write a special purpose program (in a text editing language) to create the second file from the first. Suppose that the program Edit does general text editing on one object as specified by parameters, and that file <Terminal Type Edit> contains editing instructions including IdToVec, which will perform the desired transformation. Figure 4-38 shows this third configuration.

```
subsystem TT provides TerminalTypes
    realization
        concrete object TrmTypes=file(<Terminal Type Definitions>)
                        TrmEdits=file(<Terminal Type Edit>)
        version Identifiers resources TrmTypes end Identifiers
        version Vectors
            resources Edit(TrmTypes,TrmEdits) with "IdToVec"
            end Vectors
    end TT
```

Figure 4-38: Third TT Subsystem Configuration

We can further separate the three distinct aspects of this resource. The fourth and final configuration is obtained by decomposing the information into three parts, a) the actual table values of Figure 4-29, b) the format of the values for Program1, and c) the format of the values for Program2. The symmetry of this decomposition is represented by the subsystem shown in Figure 4-39. The file <Terminal Type Definitions> contains, in canonical table format (columns separated by tab characters, for example), the information in the table in Figure 4-29. The file <Terminal Type Edit> contains editing instructions, TabToId and TabToVec, which produce, from the table values, the identifiers and the vectors, respectively.

```
subsystem TT provides TerminalTypes
    realization
        concrete object TrmTypes=file(<Terminal Type Definitions>),
                        TrmEdit=file(<Terminal Type Edit>)
        version Identifiers
            resources Edit(TrmTypes,TrmEdit) with "TabToId"
            end Identifiers
        version Vectors
            resources Edit(TrmTypes,TrmEdit) with "TabToVec"
            end Vectors
    end TT
```

Figure 4-39: Fourth TT Subsystem Configuration

We shall now add a bonus to the example. Suppose that there are two systems, with two different collections of allowed terminal types. The table of terminal types, therefore, is a resource with two possible versions. We introduce in Figure 4-40 an internal subsystem that provides one of the two sets of terminal types from a file that contains each list on a separate page.

Clearly, this process of refinement and expansion can proceed in many directions. The need for flexibility in the actual problem will dictate which of the many routes of decomposition will be followed.

4.5.2 Procedure Definition

This example illustrates the use of a subsystem to provide a program in four forms, three that can be processed by a compiler and one by a document generation program. We assume that the procedure defined here, a mathematical function, is sometimes compiled in-line and sometimes out-of-line, and that the text of the procedure is printed in the documentation using the Algol-60 reference language (using boldface and italic characters).

This program is assumed to be written in Yfpl, with a compiler that can separately compile procedures. In order to compile a procedure call, however, the procedure header and parameter specifications, bracketed by the words "External" and "lanretxE", must be inserted into the calling program. A procedure that is compiled separately, but available to other programs, must be prefixed with the word "Global". Furthermore, we assume that the Yfpl compiler also has a macro facility and that prefixing the procedure declaration with the word "Macro" causes the code to be compiled in-line, with the same semantic implications as an out-of-line invocation (although we presume that a faster implementation will result).

```
subsystem TT provides TerminalTypes requires TerminalSet
    subsystem TTS provides TerminalSet
        realization
            concrete object TrmSets=file(<Terminal Set Definitions>)
            version Set1 resources Extract(TrmSets) with 1 end Set1
            version Set2 resources Extract(TrmSets) with 2 end Set2
        end TTS
    realization
        concrete object TrmEdit=file(<Terminal Set Edit>)
        version Identifiers1 select TTS=Set1
            resources Edit(acquire(TerminalSet),TrmEdit) with "TabToId"
            end Identifiers1
        version Identifiers2 select TTS=Set2
            resources Edit(acquire(TerminalSet),TrmEdit) with "TabToId"
            end Identifiers2
        version Vectors1 select TTS=Set1
            resources Edit(acquire(TerminalSet),TrmEdit) with "TabToVec"
            end Vectors1
        version Vectors2 select TTS=Set2
            resources Edit(acquire(TerminalSet),TrmEdit) with "TabToVec"
            end Vectors2
    end TT
```

Figure 4-40: Fifth TT Subsystem Configuration

We suggest that the procedure be stored in a canonical format, such as a bracketing word "ProcedureDefinition" followed by a group of header lines, followed by a blank line, followed by the body of the procedure followed by the close bracket "noitinifeDerudecorP." An example of such a procedure is shown in Figure 4-41. (We shall ignore the presence of comments for the time being.) Let us also assume that reserved words are surrounded by apostrophes so that they can be identified for special treatment in the document processor.

```
ProcedureDefinition
'Procedure' HashFunction(Key:'String', N:'Integer');

'Begin' Hash
<< here is the body of the procedure >>
'End' Hash
noitinifeDerudecorP
```

Figure 4-41: Canonical Procedure Definition

We can easily construct four sets of editing instructions that convert the canonical form into the four desired formats. The first, which we will call ProcGenExtrn, extracts the header, deletes the apostrophes, and places the "external" brackets around it. The second editing command set, ProcGenGlobal, prefixes the procedure header with the word "Global" and deletes apostrophes from the header and body. ProcGenMacro prefixes the procedure header with the word "Macro" and deletes the apostrophes. Finally, ProcGenDoc replaces the apostrophes with special marks that cause the reserved words to be printed in boldface and all other characters in italics. All four functions are contained in the file

<Yfpl Procedure Edits>.

The subsystem in Figure 4-42 provides these procedure editing facilities. It is usable by any subsystem that wishes to define procedures in this particular manner. It is used by the subsystem in Figure 4-43 in conjuntion with example procedure from Figure 4-41. Note that resources in this subsystem include both program text pieces, such as the text that represents the particular algorithm for computing the hash function, and construction information, namely the rules for constructing the different formats of a procedure.

```
subsystem PROC provides ProcedureEdits
    realization version Yfpl resources file(<Yfpl Procedure Edits>) end Yfpl
    end PROC
```

Figure 4-42: Procedure Definition Subsystem

```
subsystem HASH provides HashFunction requires ProcedureEdits external PROC
    realization select PROC=Yfpl
        concrete object Text=file(<Hash Program Text>),
                        Edits=acquire(ProcedureEdits)
        version OutOfLine
            resources Edit(Text,Edits) with "ProcGenExtrn"
            deferred Yfpl(Edit(Text,Edits) with "ProcGenGlobal")
            end OutOfLine
        version InLine resources Edit(Text,Edits) with "ProcGenMacro" end InLine
        version Doc resources Edit(Text,Edits) with "ProcGenDoc" end Doc
    end HASH
```

Figure 4-43: Use of the Procedure Definition Subsystem

4.5.3 Systems Sharing a Subsystem

A common description problem occurs when two subsystems share a third subsystem, and it is necessary that the same version is used in both contexts. Suppose, for purposes of an example context, that a producer and consumer communicate via a queue, and that there are two versions of the queue, one using shared memory and another using a communication link. The QUEUE subsystem is shown in Figure 4-44.

```
subsystem QUEUE provides Insert, Remove
    realization
        version Memory . . . end Memory
        version Link . . . end Link
    end QUEUE
```

Figure 4-44: QUEUE--Shared Version Problem

Certainly the producer and consumer must agree on which version of the queue is selected, but no other subsystems are involved in this decision. It is natural to define a

subsystem that combines the three subsystems together and provides all the resources provided by the producer and consumer individually while delimiting the scope within which the queue is visible.

There is an approach that seems reasonable at first, but fails for reasons that will help elucidate a difficult problem in system family construction. The difficulty is in actually constructing the programs that use the resources provided by QUEUE, and therefore might need a representation for Remove or Insert. Because there are two representations for each of those resources, we cannot "use" them in constructing concrete objects for the user programs without determining the appopriate version. The skeleton shown in Figure 4-45 represents the erroneous approach to describing the system.

```
subsystem PC
    provides PrA*, PrC*, PrC*, CnC*, CnC*
    subsystem PR provides PrA, PrB, PrC
        requires Insert external QUEUE
        realization version Only . . . end Only
        end PR
    subsystem CN provides CnA, CnB
        requires Remove external QUEUE
        realization version Only . . . end Only
        end CN
    realization
        version Link select QUEUE=Link . . . end Link
        version Memory select QUEUE=Memory . . . end Memory
    end PC
```

Figure 4-45: PC--First Attempt to Share QUEUE

This seemingly reasonable configuration specifies that the selection of QUEUE is specified in the version of the system to which it applies, and that when that version is constructed, the appropriate version of QUEUE should be selected. However, when constructing pieces of PR, which actually uses the resource, it is not known in which higher level system this version of PR will be used! In fact, it could be used in several higher level system versions, each of which specifies a different selection for QUEUE. The hierarchical nature of the system descriptions causes versions of lower level systems to be independent of the locales in which they are referenced via selection clauses. In sections 4.4.2.2 and 4.4.4, we specify the restrictions on the notation that are violated above.

There are two mechanisms for properly solving the problem. One applies to the uniform use of a pervasive resource, in which case the environment structure is used. The other is more general and involves an additional construction concern.

Suppose that a producer/consumer system is built using the global queue resource, and the memory version of the queue is used uniformly throughout all versions of the system. The PRDCNSM subsystem appropriately includes the queue resources in a resource environment and specifies the version of QUEUE for all versions of PRDCNSM. This structure is given in Figure 4-46. Since the version of QUEUE is selected in the realization of PC, it is pervasive throughout PC. Hence, when PR and CN are constructed, it is completely clear what version of QUEUE to use.

This mechanism is troublesome when the variation in the versions of PC involves variations

```
subsystem PC provides PrA:*, PrB:*, PrC:*, CnA:*, CnB:*
    requires Insert, Remove environment Insert, Remove external QUEUE
    subsystem PR provides PrA, PrB, PrC requires Insert
        realization
            version Pr1 . . . end Pr1
            version Pr2 . . . end Pr2
            version Pr3 . . . end Pr3
        end PR
    subsystem CN provides CnA, CnB requires Remove
        realization
            version Cn1 . . . end Cn1
            version Cn2 . . . end Cn2
            version Cn3 . . . end Cn3
        end CN
    realization select QUEUE=Link
        version Pc13 select PR=Pr1, CN=Cn3 . . . end Pc13
        version Pc12 select PR=Pr1, CN=Cn2 . . . end Pc12
        version Pc32 select PR=Pr3, CN=Cn2 . . . end Pc32
        version Pc22 select PR=Pr2, CN=Cn2 . . . end Pc22
    end PC
```

Figure 4-46: PC--Sharing QUEUE via an Environment

in the choice of QUEUE as well as variations in the choices of PR and CN. Therefore, the resources needed from QUEUE cannot be fully determined until a version of PC is constructed. If we reconceptualize the situation, we see that it is really PC that requires and uses the queue resources, even though there are references to them in the text for PR and CN (and we hope that no significant modification must be made to PR or CN programs to accomodate this description change). The workable subsystem for this type of system is shown in Figure 4-47. The environment clause and associated global selection are gone, and PR and CN no longer require resources from QUEUE. This implies that concrete objects of PR and CN can be constructed without resolving the queue resources even though, viewed algorithmically, they use them. The concept of "use", from an algorithmic standpoint, may or may not be the same as "use", from a construction standpoint.

In effect, the components of PR and CN are skeletons of the actual program or document text. The system implementor must provide a mechanism for partially constructing the PR and CN components with the queue resources "unbound" until such a time as a PC system is built. The construction of PC will then complete the definition of PR and CN components and combine them appropriately. Note that this is not a specious inconvenience; it is necessary due to the actual difference in the content of PR and CN programs in the two PC systems.

The responsibility of the system implementor includes ensuring that the organization of the system construction steps makes the structure implied by the system description feasible. The type of flexibility required by this example has been achieved usually only within the context of macro processors that permit the definition of macros within other macros. Since there are many holes into which macro programmers fall related to evaluation and binding time, that approach is not acceptable for a reliable general purpose mechanism. Good solutions to the problem of partial program processing are yet to be developed.

Fortunately, there are several cases in which the apparent division above is in fact

```
subsystem PC provides PrA*, PrB*, PrC*, CnA*, CnB*
    requires Insert, Remove external QUEUE
    subsystem PR provides PrA, PrB, PrC
        realization
            version Pr1 . . . end Pr1
            version Pr2 . . . end Pr2
            version Pr3 . . . end Pr3
        end PR
    subsystem CN provides CnA, CnB
        realization
            version Cn1 . . . end Cn1
            version Cn2 . . . end Cn2
            version Cn3 . . . end Cn3
        end CN
    realization
        version Pc13 select PR=Pr1, CN=Cn3, QUEUE=Link . . . end Pc13
        version Pc12 select PR=Pr1, CN=Cn2, QUEUE=Memory . . . end Pc12
        version Pc32 select PR=Pr3, CN=Cn2, QUEUE=Link . . . end Pc32
        version Pc22 select PR=Pr2, CN=Cn2, QUEUE=Link . . . end Pc22
    end PC
```

Figure 4-47: PC--Successful Sharing of QUEUE

unnecessary. For example, suppose that both of the queue resource versions were implemented as externally called procedures, and therefore the actual resource representations were the same--only the deferred objects were different. We leave it to the implementor of the construction system to optimize such construction details.

4.6 Summary

In this chapter, we have presented a set of concepts useful in the representation of software systems. Some of the major features are listed below:

- Subsystems provide and require resources; those resources required by one subsystem are provided by another.

- Versions of subsystems share the interconnection structure (with possible additions specific to each version).

- The versions are organized in a tree. Leaf versions contain definitions of concrete objects that represent the resources provided by the subsystem. Non-leaf versions circumscribe information shared by a group of leaf versions.

- In the construction of any subsystem version, resources from other subsystems may be needed. The versions of the providing subsystems are specified by the requiring version.

- The construction rules for concrete system components are included in the definition of the subsystem versions. Construction rules are functional applications of programs on other concrete objects.

- Resources can be "acquired" directly in a construction rule or indirectly during the execution of a construction program.

We set out to develop a representation that integrated the subsystem integration information, an aspect of the system design, with the detailed system construction rules. Some of the examples of this chapter demonstrate that we have accomplished this task. In Chapter 5, we will sketch the design and implementation of a software construction database based on this representation, adding support to this conclusion. And in Chapter 6, we will apply this notation to an extended example, indicating the diversity of problems to which this representation applies.

5. A Software Construction Facility

The notation of Chapter 4 would be but a modest improvement over conventional methods for organizing information about software systems if there were no automated facility supporting it. In this chapter, we propose such a facility and discuss its implementation. It will not be our purpose here to describe an ultimate user facility but rather a framework within which the "ultimate" system can evolve. The process of system construction is still dominated by ad hoc configurations of processors and tools, and until substantial experience with integrated systems has been acquired, only limited design progress is possible.

5.1 Overview of the Software Construction Facility (SCF)

The software construction facility (SCF) that we will propose is organized as an interactive system that maintains a system construction database. Database entries correspond to the subsystems, versions, concrete objects and other constructs of the notation developed in Chapter 4. Programs such as compilers and editors, operating as subroutines of SCF, perform the detailed manipulations of system construction.

The user of this facility interacts with it by a) supplying the database with initial system descriptions in the notation of Chapter 4, b) modifying the database entries to reflect changes in the software configuration, and c) directing the facility to construct a software system or some component of one. A database snapshot includes the current system descriptions with annotations showing which concrete objects have been constructed. The construction history, status of each modification in progress, and usage patterns are also available to the user.

Much of the behavior of the system is defined by user specified *policies*. In addition, the user supplies special processors other than the standard, built-in processors.

Much of the implementation of this facility will be sketched by describing the abstract data types that would be involved in the implementation. The operation of commands will be defined in terms of those data types, and from this we can establish the properties of the database that the system must maintain. All of the standard database problems of synchronization, deadlock, interference, protection, backup, consistency, fault-tolerance and flaw repair apply to this database and must be solved, but are outside the scope of this thesis. An approach to the some of these problems in a software database context can be found in [Habe77].

The framework established in this chapter will not, by itself, satisfy any actual system programmer because too much information used by programmers is omitted. Some of this information, namely the matching of specifications between programming languages objects, is already being addressed by other researchers [Tich77]. Other information permits the full exploitation of the particular compilers, linkers, editors, and so forth, that are present in a given installation. Finally, in a multiproject environment, there would need to be a mechanism for providing a substructure to the facility to reflect the project domains while still permitting sharing between projects.

5.2 Description of the Database

The two repositories for information about a software system are the file system and the SCF database. Ssource information, such as programs, data collections, documentation, and processing instructions, is stored in the file system and can be manipulated in the ways commonly used by programmers. The SCF database contains subsystem descriptions,

construction processes, and construction histories. Some of the database information is provided directly by the user, some of it arises from software construction operations, and the remainder is generated by the software construction policy enforcer.

5.2.1 Low Level Database Types

The following data type definitions are provided here for reference during the discussion of the more complex data types. Part of the reason for presenting them is to introduce the programming notation that will be used in this chapter.

The notation we will use to describe these program segments is derived from current programming language research. Unfortunately there is no single language with the sufficient currency to be the prototype for such a notation, in the manner that Algol-60 served for the "Algol-like" languages. The list module in Figure 5-1 will serve to illustrate, in a context with which we are all familiar, the notational conventions used in this thesis.

Two possibly unfamiliar notations are used. The first, an "&", indicates a type argument to a generic type definition. For example, the definition of type "*Stack* of *&T*" permits the declaration of "*Stack* of *Integer*" or "*Stack* of *PersonnelRecord*." The other notation is an exceptional exit; for our purposes it need only have one or two values, called here *succeed* and *fail*. A routine can also be declared to either return a value or *fail*, meaning that either it computes a value or it raises an exception and has no value. An *on* clause specifies, for a block, the action to take if an exception is raised within the block. Unless otherwise specified, a *fail* exception raised within a block causes that block to raise a *fail* exception. Other possibilities include providing a value for a failed block (e.g. "on fail use NIL") and providing a value for a block in which an exception is raised (e.g. "on fail return(false)"). The *List* type, given in Figure 5-1, provides simple lists with uniformly-typed elements; i.e. the list element values may be of any type, but all elements of a given list are of the same type. There are operations that scan, search and merge lists, and that apply functions to the list element values. Some lists are used to implement sets and stacks, and therefore set and stack operations are also defined.

Many of the transactions on the database are recorded by *events*. These events will be defined very simply in Figure 5-2 although the structure of an event in a real system would be more complex.

```
type Event
   exports When: Timestamp, What: String, Details: List of String
   end Event
```

Figure 5-2: Definition of Type *Event*

Events are entered into *histories* attached to each database entry and are included in *messages* (see Figure 5-3) sent to database entries. For example, if a change is made to a file that is used in the construction of concrete object x that is a resource definition for version y of subsystem z, an event is entered into the history attached to x and messages containing the event are sent to each of y and z recording the change.

The construction *rules* are also database objects; the type definition is given in Figure 5-4. Each rule specifies a processing program, a list of concrete object parameters and a list of string parameters.

```
type List of &T                          ! define List type
  type ListNode
      exports Value: &T, Next: ref(ListNode)
      end ListNode
  var First: ref(ListNode) initially NIL
      Last: ref(ListNode) initially NIL
      Count: Integer initially 0
  comment list-style operations;
  let (L,L1,L2) be List, E be &T
  operations
      NewList of &T returns List         ! creates a new list header
      Clear(L)                           ! deletes all elements of list
      Append(L,E);                       ! appends element to list;
      Lop(L) returns &T or fails         ! truncates head and returns value
      Find(L,F) returns &T or fails      ! returns first entry E satisfying F(E)
      Apply(L,F) returns List            ! creates list of F(E) for each E in L
      Concat(L1,L2)                      ! attaches list2 to end of list1
  comment set-style operations
  let (S,S1,S2) be List, E be &T
  operations
      Empty(S) returns Boolean           ! true if set has no members
      Insert(S,E)                        ! insert element into set
      Remove(S,E)                        ! remove element from set
      Union(S1,S2) returns List          ! creates new set from union of sets
      Intersect(S1,S2) returns List      ! creates new set from intersection of sets
      Merge(S1,S2)                       ! merges elements of S2 into S1
      Member(S,E) returns Boolean        ! true if element is a member of the set
      Subset(S1,S2) returns Boolean      ! true if S1 a subset of S2
  comment stack-style operations
  let S be List, E be &T
  operations
      Push(S,E)                          ! pushes E onto stack
      Pop(S) returns &T                  ! pop element from stack

  end List
```

Figure 5-1: Definition of Type *List*

```
type Message
  exports Code: String, Reason: ref(Event)
  end Message
```

Figure 5-3: Definition of Type *Message*

```
type Rule
  exports Processor: <processor description>
      ConcreteParms: List of ref(ConcreteObject)
      StringParms: List of String
  end Rule
```

Figure 5-4: Definition of Type *Rule*

A *selection* specifies an association between a subsystem and a versions of that subsystem. The specified version is always a leaf of the version tree. This type is given in Figure 5-5.

```
type Selection
  exports Subsystem: ref(Subsystem)
      Version: ref(Version)
  end Selection
```

Figure 5-5: Definition of Type *Selection*

A *policy* is a condition and a set of actions, as shown in Figure 5-6. The condition is a boolean expression on the database entries and the actions are SCF commands. Policies are used to enforce consistency in the database and inform users of modifications to database entries.

```
type Policy
  exports Conditions: <Boolean Expression>
      Actions: List of <Command>
  end Policy
```

Figure 5-6: Definition of Type *Policy*

5.2.2 High Level Database Types

Each of the major constructs of the system structuring notation has an analogous entry type in the database. The translation is straightforward; lists of *xxx* in the notation become *Lists* of *xxx* in the database entry. In addition, each entry has structures used by SCF in the implementation of construction processes, mailboxes, and histories.

The *subsystem* type, shown in Figure 5-7, exports, in addition to the visible values such as the version list of the subsystem, as set of standard operations and a set of special operations. The standard operations are defined in all high level types and may be invoked by the command language of SCF to effect a user request to create, modify, display or destroy an instantiation of the type. The special functions are used internally by SCF and include operations to perform the mappings implied by the subsystem description (e.g. resource to provider, subsystem to version).

The *Version* type in Figure 5-8 has, in addition to the standard operations, special

```
type Subsystem
  exports Name: String
      Requires,Provides,Environment: List of Resource
      Nested: List of Subsystem
      Externals: List of ref Subsystem
      Objects: List of ConcreteObject
      Selections: List of Selection
      Versions: List of Version
      Policies: List of Policy
  internal Context: ref(Subsystem)          ! Enclosing Subsystem (if any)
      Usage: List of ref Subsystem          ! Subsystems that use this subsystem
      History: List of Event                ! Complete history of this Subsystem
      Mail: List of Message                 ! Messages to this Subsystem
  let sbs be Subsystem, rsrc be Resource, target be ref(Subsystem)
  operations
      NewSubsystem returns Subsystem! creates a new subsystem
      Display(sbs)                          ! displays a subsystem
      Modify(sbs)                           ! modify for description editor
      Destroy(sbs)                          ! destroy a subsystem
      FindSelect(sbs,target) returns ref(Version) or fails
          comment FindSelect determines which version of the target
              subsystem has been selected in the realization section of sbs
          define Test(x): Subsystem(x)=target
          return(Version(Find(Selections(sbs),Test)))
          end FindSelect
      FindExternalProvider(rsrc,sbs) returns ref(Subsystem) or fails
          comment FindExternalProvider finds which subsystem in the
              externals list of sbs provides the resource rsrc
          define Test1(x): begin on fail return(false)
                      define Test2(y): y=rsrc
                      Find(Provides(x↑),Test2) true end
          return(Find(Externals(sbs),Test1))
          end FindExternalProvider
      FindNestedProvider(rsrc,sbs) returns ref(Subsystem) or fails
          comment FindNestedProvider finds which subsystem in the
              nested list of sbs provides the resource rsrc
          define Test1(x): begin on fail return(false)
                      define Test2(y): y=rsrc
                      Find(Provides(x),Test2) true end
          return(Find(Nested(sbs),Test1))
          end FindNestedProvider
  end Subsystem
```

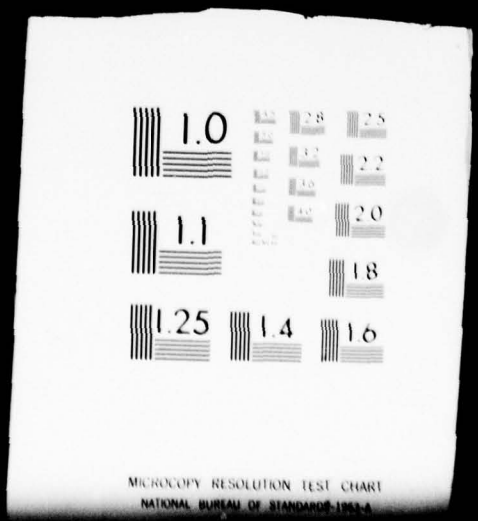
Figure 5-7: Definition of Type *Subsystem*

operations to perform the mappings of subsystem names to versions, resources to concrete objects, and resources to internal providing subsystems.

The *Concrete Object* type in Figure 5-9 has only the standard high level object operations. However, the *Construct* operations, discussed in section 5.4.2, can be considered an operation on this type.

AD-A070 955 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE REPRESENTATION OF FAMILIES OF SOFTWARE SYSTEMS.(U)
APR 79 L W COOPRIDER F44620-73-C-0074
UNCLASSIFIED CMU-CS-79-116 AFOSR-TR-79-0732 NL

2 OF 3
AD A070955



```
type Version
  exports Name: String
      Selections: List of Selection
      Objects: List of ConcreteObject
      Subsystems: List of Subsystem
      Versions: List of Version
      Components,Resources,Deferred: List of ref ConcreteObject
      Policies: List of Policy
  internal Scontext: ref(Subsystem)      ! Subsystem of which this is a version
      Vcontext: ref(Version)             ! Version of which this is a version (if any)
      History: List of Event
      Mail: List of Message
  let ver be Version, rsrc be Resource, target be ref(Subsystem)
  operations
      NewVersion returns Version         ! creator
      Destroy(ver)                       ! destroy a version
      Display(ver)                       ! display version definition
      Modify(ver)                        ! for the description editor
      FindSelect(ver,target) returns ref(Version) or fails
          comment FindSelect determines which version of the target
              subsystem is specified in the version selection clause
          define Test(x): Subsystem(x)=target
          return(Version(Find(Selections(ver),Test)))
          end FindSelect
      FindResource(ver,rsrc) returns ref(ConcreteObject) or fails
          comment FindResource gets the concrete object which is
              associated with rsrc in the "resources" list of ver
          define Test(x): x=rsrc
          return(Find(Resources(ver),Test))
          end FindResource
      FindDeferred(ver,rsrc) returns ref(ConcreteObject) or fails
          comment FindDeferred gets the concrete object which is
              associated with rsrc in the "deferred" list of ver
          define Test(x): x=rsrc
          return(Find(Deferred(ver),Test))
          end FindDeferred
      FindProvider(rsrc,ver) returns ref(Subsystem) or fails
          comment FindProvider finds a subsystem within ver
              which provides rsrc
  end Version
```

Figure 5-8: Definition of Type *Version*

```
type ConcreteObject
  exports Name: String
      Construction: Rule
      Policies: List of Policy
  internal History: List of Event
      Mail: List of Message
      Scontext: ref(Subsystem)          ! Subsystem containing definition
      Vcontext: ref(Version)            ! Version containing definition (if any)
      Location: File or Temporary       ! Contents of ConcreteObject
      Usage: List of ref(ConcreteObject)
                                        ! ConcreteObjects using this
      Composition: List of ref(ConcreteObject)
                                        ! ConcreteObjects used by this
      Deferred: List of ref(ConcreteObject)
                                        ! ConcreteObjects attached to this
  operations
      NewConc returns ConcreteObject ! creator
      Modify(conc)                      ! description editor
      Destroy(conc)                     ! destroy a ConcObj
      Display(conc)                     ! display a ConcObj
  end ConcreteObject
```

Figure 5-9: Definition of Type *Concrete Object*

The type *File*, shown in Figure 5-10, is a database entry that maintains the status of a file within the file system. Assuming that all access to such files is done via SCF, the standard high level type information is available about them.

```
type File
  exports Path: <catalog entry>     ! File system catalog path
      Policies: List of Policy
  internal History: List of Event
      Mail: List of Message
      Level: IntegerPair            ! Update level indication
      EditControl: < . . . >        ! Locks, authorizations, etc.
  end file
```

Figure 5-10: Definition of Type *File*

5.2.3 State of the Database

We view the software construction activity as a continuous, dynamic activity. At any time, different versions of each system may be undergoing changes, possibly at a rapid rate. Since substantial computing is required in order to construct a system, changes to the data base cannot be considered to be instantaneous; there will often be a set of construction processes in progress. Some of the processes are the direct result of the requests of the (potentially numerous) users, but others are the spontaneous outgrowth of user defined policies.

The dynamic state of the database is maintained by SCF using a *to do list* (a queue of construction processes) and a communication mechanism in which *events* are recorded in *histories* and *mailboxes* that are attached to major database entries. These state variables may be interrogated directly by the user or indirectly with user-defined policies.

5.2.3.1 Histories. The history of each entry is recorded in the database as a sequence of events. For all entries, the history shows when the entry was created, how and when it was modified, and how and when it was involved in construction processes. For a *file* entry, it also indicates when and by whom the contents of the file was modified (and in what manner, if that information is available).

Histories of this kind are typically large and cannot be stored online forever. Therefore, policies refer only to some recent subset of the history, perhaps some number of entries for "usage" history and a similar number of entries for "change" history. Some historical information is summarized for policy use; for example, the list of users of a concrete object is recorded in the corresponding entry even though the history for the usage events has been migrated to the tape library.

5.2.3.2 Mailboxes. Each event is packaged in a message and sent to each database entry that might be affected by the event. SCF initiates some messages, such as file modification, database entry modification, and "alarm clock" messages. In addition, user policies generate messages as one result of receiving messages.

The mailbox of an entry is managed by the user and the policies associated with the entry. It is intended that mailboxes, unlike histories, remain small and continuously avaiable for interrogation. Policies might dispose of most messages immediately, but users may need to dispose of unusual messages or handle difficult situations.

By having "human being" entries in the database, the user can specify that SCF send messages to individual people under certain circumstances. Such a scheme could minimize the database monitoring necessary by, for example, sending a message to the subsystem manager when a version loses its last user or the size of its mailbox exceeds 50 messages.

5.2.3.3 To Do List. User commands and policy actions enter construction processes in a [to do list. Most of these processes are independent of each other and can be executed simultaneously. On occasion, however, a construction step may need a concrete object for which an entry is already in the to do list; that step must wait.

SCF checks each completed construction step and, if it is successful, activates any construction processes awaiting it, sends a "construction complete" message to the database entry for the object and may send a "component change" message to users of the resulting object. If a construction step fails, e.g. a compiler detects a syntax error, a "construction failure" message is sent to entries for the object itself and any waiting objects.

5.3 Command Language

The user of this system has a command language with which to direct the behavior of the system. It is not within the scope of this thesis to define a particular command language; command languages should be compatible with similar languages available to users of the operating system and should reflect local user style. It could be a simple command/operand

imperative language or a general expression evaluation language; it might require specification of parameters on the command line, or prompt for them individually. This section discusses the functionality that the command language must provide.

5.3.1 Interrogation

Users interrogate the database to examine the actual entries, predict SCF behavior, and extract data. The interrogation commands for these operations use differing amounts of information about the database and its manager.

In display mode, the user examines individual entries in the database (possibly with subentries expanded). For example, the user might display a version entry to determine what concrete objects constitute its resources, or a concrete object to determine if it was original source or generated text. These commands use only the type definitions of the database entries.

Resolution mode commands exploit an understanding of the SCF operation and the semantics of the *acquire* function. Such a command could list the complete collection of versions or concrete objects that would be used by a particular construction process, or explicitly name the origin of all resource requests in a subsystem.

Other interrogations are built on a base of primitives combined with a query language. Operations to extract history and mailbox entries and to ennumerate various types of objects permit the user to answer such questions as

- What construction was performed between Monday and Wednesday?
- In what contexts is the YFPL compiler used?
- What files have not been used for a year?
- When did Fred last modify the system?
- What did Fred do to the system last week?

5.3.2 Construction

We use the term "construct" to mean the invocation of processes that create a particular concrete object. These processes may be the null set of processes (if the object already exists) or may be a lattice of processes terminating in the creation of the concrete object. The term "build", on the other hand, is defined to mean to invocation of the set of processed that cause the construction of all concrete objects in the component list of a version and of the selected versions of all internal subsystems.

The user of SCF normally specifies what system versions are to be "built", letting SCF infer the concrete object constructions that are necessary. However, the explicit construct command provides additional control over the sequence of construction processes for testing or optimization.

We make it an SCF policy that a resource can be used only if the version that supplies the resource has been built. This is necessary if we want any constructed program to execute successfully without additional construction commands being necessary "on the side."

Therefore, not only do "build" commands cause the construction of concrete objects, but "construct" commands can cause the building of versions.

5.3.3 Entry Editor

Subsystem descriptions entered into the database are, like all aspects of a system, likely to change. These changes reflect new system design, improved construction techniques, or additional variants. The database entry editor provides a mechanism for selectively modifying the contents of the database.

The editor does not operate on the textual representation of the system description but on the typed database objects themselves. Each database type provides a Modify operation that permits the user to change field values, add or delete elements from lists, and to modify list elements (thereby invoking the Modify command from the type of the list element). Objects contained within other objects (as opposed to referenced by them) cannot be modified without modifying the containing object. Conflict between construction and modification can be prevented with synchronization techniques on such a hierarchical database.

Entry editing results only in local modifications in the database. However, many non-local objects constructed with resources from the modified subsystem are potentially incompatible with the new description. Modification of a subsystem description causes user subsystems to be notified in a manner similar to that caused by modification of a source file. See section 5.3.4 for the effects of modifications on the database.

5.3.4 Policies

Often, if a source file for a system version is modified, the system version itself must be rebuilt. However, it is not the case that system implementors generally wish those modifications to occur immediately. Suppose that there are several versions of a system active, all executable; one is available to customers, one is backup, and five are for development by five programmers. The project leadership might determine that no modification should ever be propagated into the backup version, while modifications may be installed in the customer version on explicit request. Of the five programmers, two may permit modifications to their test systems whenever any programmer changes any subsystem, while another may permit only overnight upgrades and the other two wish to have complete control over modifications to their test systems.

Even more complex conditions might govern the non-executable portions of systems. Consider the documentation files of a system, some of which are online, and others of which are printed on paper and distributed. The former should be forced to change in parallel with the executable program available to users, while the latter need only be upgraded when a documentation update is produced.

For another example, consider the distribution tape sent to customers who install the system at their installations. This tape might contain a frozen version of the system, with a set of comments about recent modifications; this would reflect a conservative approach with high priority on uniform distribution. Alternatively, it could be regenerated each time a tape was created, causing each customer to get the "most recent" edition of the system. A third possibility would be to incorporate in the distribution tape any modification that had been installed on the local system for at least thirty days.

In short, the user must be able to specify policies that control the propagation of modifications thoughout the entire database, and many of those policies are specific to

particular versions of particular systems. SCF provides a mechanism for enforcing policies specified by the users.

Policies are attached to objects within the database or to a limited number of special objects, such as a clock. These policies are able to detect certain conditions in the database. A completely general mechanism for determining conditions within a database is, of course, extremely expensive, so our goal will be to capture as many of the types of conditions that are needed as possible within the framework of a simple, efficient mechanism.

We select a notation for policies that is known to be general; efficiency will therefore derive from restrictions on the timing of policy usage rather than the scope of possible policies. Each policy shall be described by a set of productions of the form

condition → action

where conditions are predicates on elements of the database and actions are sequences of command language statements. We shall use prose conditions and actions in order to avoid excessive concern with syntax.

Conditions include tests on mailbox entries, history files of the associated object, and possibly history files of other objects. Suppose that version A of subsystem B has a concrete object C that uses a concrete object that has been modified. As described in section 5.2.3, a "component change" message would be sent to the mailbox associated with C. A simple policy that specified immediate reconstruction of this object upon any modification would be written

upon receipt of a "component change" message, construct C

To specify conditions, a set of operations must be defined on histories, events and mailboxes. The primitive operations permit extraction of entries according to various conditions (e.g. age of entry, identity of programmer) and use of the contents of entries (e.g. testing the type of message). We assume a reasonable set of such primitives in the examples below.

The action clauses differ from command strings issued by the user only in that they are executed automatically when conditions arise. They may operate concurrently with each other and with user command sequences.

We limit the time necessary to evaluate the policies by doing evaluations of conditions only at times when it is likely that some conditions could become true, or when consistency is particularly relevant. Most often, conditions become true when a particular event occurs within the scope of the database. By requiring that events send messages to objects that might be affected, we can operate only on the policies associated with objects that have received messages. Some of these messages will be sent automatically, whereas others will be the result of policies within the originating object.

An automatically generated message would result, for example, from the modification of a file. Suppose that a file is submitted, under system control, to an editor so that a user can make modifications to it. When that file is returned to the environment after editing, messages are sent to all objects that were constructed using that file and to the version and subsystem in which it was defined. Policies associated with each of those objects would further refine the meaning of the event, and possibly result in a set of actions to be performed by the system.

As an example of a policy generated message, consider the online system command

descriptions that change only when the current user system changes. In this case, a policy within the current user system object would specify that a message should be sent to the online command description object whenever a modification to the system occurs. Within the online command description object, a policy would specify that if a message is received dictating an update, and a message already is present indicating that a component change has occurred, then the online command description object should be copied to a backup and recreated.

Some conditions within the database will become true due to the passage of time. Hence, a special object within the system, a clock, can also be programmed to send messages to objects that have time dependent policies.

Another special time to check policies is at the time that the object is used. If no other trigger has caused evaluation of the policy conditions, usage of the object can be made conditional on the satisfaction of a policy. For rarely used objects, this facility can be used to recreate only those objects that are actually used, even though many more may have been affected.

Combinations of policies can be used to accomplish some optimizations. Suppose that object x is derived from object y, and is used in object z. When object y changes, object x must be recreated, and normally so would object z. But it might be the case that object x is identical before and after the change to y. If y is a procedure definition, and x is the header extracted for external linkage, a change to the body of the procedure does not change the header. There is certainly no need to regenerate z since no constituent of z has actually changed. A string comparison program can provide a predicate that permits the policy associated with x to terminate the propagation of the modification.

- Policy(y) - *upon modification, send "component change" message to all objects generated from y.*

- Policy(x) - *upon receipt of "component change" message, copy x and regenerate x. If new x differs from old x, send "component change" message to all objects generated from x.*

- Policy(z) - *upon receipt of "component change" message, regenerate z.*

5.4 Central Facilities Implementations

It would not be appropriate to outline the entire implementation of SCF in this thesis. Much of the implementation is obvious, other aspects are unimportant, and still other parts are arbitrary and could easily be replaced by other, equally valid, implementations. However, a few of the facilities are truly fundamental to the concepts that this thesis addresses, and we can clarify them by discussing implementation in detail.

The following sections present details of the implementation of the *acquire* algorithm, which obtains that representation of a resource that is appropriate for the environment from which it was requested, and the *construct* driver, which causes the construction of a concrete object. The programs have been simplified to assume a sequential SCF construction process manager such that the *to do list* is subsumed by the SCF call stack.

5.4.1 Acquire

The *acquire* mechanism "resolves" resource names by obtaining representations of the resources. It may be invoked directly in the definition of an object (see section 4.4.4), or it may be invoked during the operation of some other processor, such as a compiler or editor. In either case, *acquire* is always invoked as an aspect of the construction of some concrete object. Therefore, the complete construction context for an *acquire* invocation is always known *a priori*; we consider it a value global to the invocation of acquire, called *AcquireContext*. The resource that *acquire* has been called to resolve is a parameter to the invocation.

Here is the set of conditions necessary for the success of *acquire*:

- The resource must be present in the require list of the subsystem within which the object is defined (the current subsystem).

- The resource must either be present in the provides list of a subsystem immediately nested within the current subsystem, or in the provides list of a subsystem referenced in the externals clause of the current subsystem, or in the environment clause of a subsystem that textually encloses the current subsystem.

- The version selector must be available within the scope that contains the object being constructed. In the case of objects defined within a realization section, the selector may be in the realization section of the current subsystem. For objects defined in a version, the selection may also be specified in any version which includes the version in which the object is defined. Finally, if the resource is a part of a resource environment, the selection is provided in the subsystem that establishes the environment.

By definition, *acquire* obtains a resource in its final form. It may not be able to do so by finding, according to the above criteria, the subsystem that immediately provides the resource, because that subsystem may in turn require it from some other subsystem. In such a case, *acquire* must repeat the search until it arrives at the eventual providing subsystem. At each step, the proper version is determined simultaneously, and therefore the correct version of the eventual providing subsystem can be identified. All that remains is to look up the resource in the *resources* list of the version at hand, and produce the object that corresponds to it.

That object, however, may not have been constructed or may have become obsolete; in this case *Construct* is recursively invoked to construct the object. As a result, *acquire* might also be recursively invoked to resolve resources for this new construction.

When the construction is successfully completed, *Acquire* returns the string contained in the target object to the calling site (within AcquireContext). In addition, deferred objects corresponding to this resource are merged into the deferred objects list of AcquireContext. Finally, history entries are made to record the usage of the new object by the one being constructed.

The algorithm for *Acquire* is outlined in Figure 5-11. The loop body dictates the priority of providing subsystems. If we permit ambiguity about the origin of resources (and this is not necessarily a good idea) then we should establish how to pick the correct subsystems. The

following list shows the priorities implemented in Figure 5-12:

1. internal version subsystems
2. internal subsystems
3. external subsystems
4. environment subsystems

Within any group of subsystems, the most closely nested subsystems has the highest priority. Within a nesting level, the first subsystem listed has highest priority.

```
global var AcquireContext: ref(ConcreteObject)

procedure Acquire(rsrc: Resource) returns String or fails
    var sbs: ref(Subsystem) initially NIL
        ver: ref(Version) initially NIL
        conc: ref(ConcreteObject) initially NIL
    sbs ← Scontext(AcquireContext↑)
    ver ← Vcontext(AcquireContext↑)
    repeat <set "sbs" and "ver" to the providing subsystem and version>
        until <"rsrc" not required by "sbs/ver">
    conc ← FindResource(ver↑,rsrc)
    <enforce "conc" policies keyed to usage>
    Construct(conc); Build(ver)
    <side effects of Acquire>
    return(Contents(Location(conc↑)))
    end Acquire
```

Figure 5-11: Acquire--Top Level Algorithm

```
repeat
        if not <found in version subsystems>
        thenif not <found in nested or external subsystems>
        thenif not <found in environments>
        then fail fi
    until <"rsrc" not required by "sbs/ver">
```

Figure 5-12: Acquire--Loop Body Refinement

There are two cases for which a resource is required by a version of a subsystem: a) it is named in the requires list of the subsystem or b) it is named in the provides list of a subsystem within the version. Therefore the program in Figure 5-13 implements the test at the end of the loop.

The <found in *xxxxx*> clauses can be considered boolean expressions with side effects on *sbs* and *ver*. If we presume some simple list manipulation procedures for handling the substructures of the subsystems and versions, the code sections in Figures 5-14 to 5-16 will

```
<"rsrc" required by "sbs/ver">

define ReqLook(y): on fail use NIL
        define ReqLook(x): <"rsrc" in provides lists of x↑>
        return(Find(Subsystems(ver↑),ReqLook)≠NIL or
                <"rsrc" in requires lists of sbs↑>)
```

Figure 5-13: Acquire--Loop Test Refinement

suffice for the loop body pieces.

```
<found in version subsystems>

var s: ref(Subsystem), v,hv,tv: ref(Version) initially NIL
on fail use NIL;                          ! failed routines return NIL
if ver=NIL then return(false)
   else v ← ver
      repeat s ← FindProvider(rsrc,ver↑)
             hv ← v; v ← Vcontext(v)
         until s≠NIL or v=NIL
      if s=NIL then return(false)
         else v ← ver; tv ← FindSelect(s,ver↑)
            while tv=NIL and v≠hv do
               v ← Vcontext(v); tv ← FindSelect(s,v↑) od
            if tv=NIL then fail else sbs ← s; ver ← tv; return(true) fi fi fi
```

Figure 5-14: Acquire--Found in Version Subsystems Refinement

```
<found in nested or external subsystems>

var v,tv: ref(Version), s: ref(Subsystem)
on fail use NIL
s ← FindNestedProvider(rsrc,sbs↑)
if s=NIL then s ← FindExternalProvider(rsrc,sbs↑) fi
if s=NIL then return(false)
   else v ← ver; tv ← FindSelect(s,ver↑)
      while tv=NIL and v≠NIL do
         v ← Vcontext(v↑)
         if v≠NIL then tv ← FindSelect(v↑,s) fi od
      if tv=NIL then tv ← FindSelect(sbs↑,s) fi
      if tv=NIL then fail else sbs ← s; ver ← tv; return(true) fi fi
```

Figure 5-15: Acquire--Found in ... Subsystems Refinement

The boolean expressions used above are expanded in Figures 5-17 to 5-19 for completeness. The expression <resource in *xxx* list> indicates that the resource can be provided from the resource in the specified list.

```
<found in environment>

var v: ref(Version), s: ref(Subsystem)
on fail return(false)
s ← sbs
while not <resource in environment of s↑> and Context(s↑)≠NIL do
    s ← Context(s↑) od
if <rsrc not in environment of s↑> then fail fi
sbs ← s; ver ← NIL; return(true)
```

Figure 5-16: Acquire--Found in Environment Refinement

```
<rsrc in provides list of sbs↑>

on fail return(false)
define Test(x): x=rsrc
Find(Provides(sbs↑),Test); return(true)
```

Figure 5-17: Acquire--Provides Predicate Refinement

```
<rsrc in requires list of sbs↑>

on fail return(false)
define Test(x): x=rsrc
Find(Requires(sbs↑),Test); return(true)
```

Figure 5-18: Acquire--Requires Predicate Refinement

```
<rsrc in environment of s↑>

on fail return(false)
define Test(x): x=rsrc
Find(Environment(s↑),Test); return(true)
```

Figure 5-19: Acquire--Environment Predicate Refinement

Referring again to Figure 5-11, the Build procedure constructs each of the components of the version that supplied the resource. The details of policy enforcement and history for Build are omitted in Figure 5-20; they are similar to those of *Construct* (see section 5.4.2). Finally, we present in Figure 5-21 the definitions of the side effects of *Acquire*.

5.4.2 Construct

The operation *Construct* invokes all processes necessary to construct some one specific concrete object. It can be invoked by the user via the command language, by the system as a result of enforcing a policy, or as a side effect of another construction either directly (as when an input object must be constructed), or indirectly (as during the operation of *Acquire*).

```
Build Procedure Skeleton

procedure Build(ver: ref(Version)) succeeds or fails
    ...
    Apply(Components(ver↑),Construct)
    ...
    end Build
```

Figure 5-20: Acquire--Relevant Sections of Build

```
<side effects of Acquire>

Insert(Usage(conc↑),AcquireContext)
Insert(Composition(AcquireContext↑),conc)
Merge(Deferred(AcquireContext↑),Deferred(conc↑))
Insert(History(conc↑),
    Event(<timestamp>,"used as resource",
        Concat(Append(NewList of String,Name(AcquireContext↑),Name(rsrc)),
                SubRsrcs(rsrc))))
```

Figure 5-21: Acquire--Side Effects Refinement

The system alternates between *Construct* and *Acquire* during the construction of any complex system component.

Construct is a function whose only parameter is the name of a concrete object. Since each concrete object is defined within the context of a subsystem and (optionally) a version, the context within which the concrete object is to be built is unambiguously identified; in fact, it is that disparity in context that distinguishes two concrete objects that are constructed using the same rule on the same input objects.

Acquire always operates as a subroutine for *Construct*. The parameter to *Construct* establishes the global context within which *Acquire* is called. This is more than just a linguistic concern, because the *Acquire* call may come from within a compiler or editor. Constructions necessary to complete the *Acquire* function will be for different objects, so the context established for *Acquire* must be stacked at each entry to *Construct*. We will explicitly portray this in the programs in Figure 5-22 to emphasize the relationship between these two central facilities.

Construct also performs the housekeeping operations necessary to maintain the validity of the database. These include deleting previous usage links, entering history information, and notifying current users of the modification.

The optimization test first determines if a valid copy of the object is already available. If it does, it is either in a file in a temporary object (which might also be a file). If not, it may be because it has never been built, or because the user destroyed it, or SCF destroyed it as the result of a policy, or because the file system destroyed it due to an error or a management policy. (In systems where purges or migrations occur as a matter of course, an interface between SCF and the file system can greatly enhance the effectiveness of the

```
var ACStack: Stack of ref(ConcreteObject)

procedure Construct(conc: ref(ConcreteObject)) succeeds or fails
    if Location(conc↑)=NIL
        then Push(ACStack,AcquireContext)
            AcquireContext←conc
            <preliminary housekeeping>
            <invoke processor on parameters>
            AcquireContext←Pop(ACStack)
            if <processor successful>
                then <success housekeeping>
                else <failure housekeeping> fi fi
    end Construct
```

Figure 5-22: Construct--Top Level Algorithm

purging or migrating facility.)

The "processor" named in the rule for the construction of the object is either a special processor or a general processor. The special processors are *acquire*, *deferred* and *file*; all other processors are programs generally available to users of the installation or special purpose processors built by individual projects or programmers.

The special processor *file* determines that the named file exists. If necessary, authorization is established, validity is checked, and any other properties deemed necessary to reliable operation of the system are established. No actual "construction" is performed unless special formatting is necessary. For example, if the editor maintains files in an "original plus delta" format, so that the previous copies of the file are maintained with the current copy, the current copy must be extracted before being passed on to a compiler.

The special processor *acquire* creates a concrete object from a resource. Depending on the representation of concrete objects, this may be as simple as pointing at the file that contains the resource object, or as complex as converting from a segment to a file, or a file to a string. The *acquire* processor is, of course, implemented by an internal call on *Acquire*, which is described in section 5.4.1.

The *deferred* special processor makes a new concrete object from those which have accumulated in the *deferred* list of another concrete object.

General processors are such programs as compilers, editors, linkers, document production programs, database processing programs, and special purpose programs such as BNF-to-parser converters, finite-state-machine-to-program converters, catalog display programs, cross reference programs, test programs, flowchart generators, and other tools for manipulating the objects that constitute systems. The interfaces to these programs are highly idiosynchratic, and therefore must be provided in a processor definition table maintained by the SCF user. The parameters in this table include the input files and a set of processor parameters. The interface prepares an invocation sequence appropriate for the processor and invokes the processor in such a way that the state of the database is entirely maintained. Remember that during these general processor executions *Acquire* will be invoked, and the context must be known to SCF at that time.

```
<preliminary housekeeping>

define Purge(x): Remove(Usage(x↑),conc)
Apply(Composition(conc↑),Purge)
Clear(Composition(conc↑))
Clear(Deferred(conc↑))

<success housekeeping>

define Send(x): SendMail(x↑,NewMessage("component change",@e))
e ← NewEvent(<timestamp>,"constructed", . . .)
Append(History(conc↑),e)
SendMail(conc↑,NewMessage("history entry",@e))
Apply(Usage(conc↑),Send)

<failure housekeeping>

e ← NewEvent(<timestamp>,"construction fail", . . .)
Append(History(conc↑),e)
SendMail(conc↑,NewMessage("history entry",@e))
```

Figure 5-23: Construct--Housekeeping Refinement

5.5 Summary

In this chapter we have explored the representation proposed in Chapter 4 by designing a software construction facility around those concepts. By exploring, albeit cursorily, the basic design of several aspects of the facility, its feasibility has become clear. We presented the implementation of those portions of SCF that are both complex and central to the usefulness of our particular representation.

Several parts of this system have been implemented and run on a PDP-10 computing system. Although a consistent version of all of these parts has never existed simultaneously, we have tested various versions of the acquire function, a language processor, a database entry editor, the construct function and the message sending mechanism. Major aspects discussed in this chapter but not implemented include the policy enforcer and the control of concurrent actions in the database.

III Discussion

6. Example Target System

The concepts presented in the previous chapters are intended for use in the representation and processing of real families of real systems. In order to explore the application of the concepts, and demonstrate their feasibility, we will present the representation of portions of a real system in sufficient depth to understand the details of the system construction process for this system.

There are multiple goals for this chapter. First, we hope to substantiate the claim that the concepts are generally useful in application to complex systems. Second, by treating the system in depth, we shall provide additional detailed information not necessarily apparent from the general descriptions in Chapter 4. Finally, we shall determine some of the limitations of the concepts as presented, and find problems for which better solutions are necessary.

The target system is not a toy system. We will discuss the target system enough to make the representation meaningful, justify the choice of the target system, and then represent significant portions of it.

6.1 Description of the Target System

The target system is the software support for a scanline graphics printer*. The printer is capable of printing arbitrary graphics on continuous eight-and-one-half inch paper, placing black points at the resolution of 200 points per inch. The graphic pattern for text information is determined by using the ASCII character codes to look up rectangular grids of points in a "character set" definition. Less structured information is displayed by supplying the actual bit patterns to be printed ("graphics" mode).

The printer is driven by a dedicated minicomputer that is a slave to a general purpose timesharing system and is connected to it by a communication link. Users on three other computing systems access the printer via a network. The software support for this system consists of the printer driver programs interfaces for a variety of other programs.

6.1.1 Purpose of Target System

The target system provides five primary services for the community of users. The central facility is the ability to print formatted documents. The other facilities include the users' manual, the interfaces for document preparation systems, character set manipulation facilities, and interfaces for programs which process printable documents. These will be discussed in turn in the sections which follow.

Driver Programs Users print documents on the scanline printer with the assistance of a driver program set. There are two kinds of documents, text-oriented documents and graphics-oriented documents. The text documents consist primarily of characters, although there are often control codes embedded in the stream to invoke features such as character set selection, paper cutting, spacing, headings, justification, underlining, overstriking, tabulating and subscripting. In addition, a graphics-oriented document can be embedded in a

*The text you are reading is an example of its ouput.

text-oriented document, as when figures are placed in running text.* Graphics documents consist mostly of scanline descriptions, with occasional control codes that allow cuts and blank lines. The scanline descriptions themselves are encoded either as bit strings or run codes (alternating counts of black and white bits), or a combination of the two; they are encoded in the format interpreted by the printer hardware.

The driver is a pair of programs. One program runs on the minicomputer and is responsible for controlling the printer, translating character codes to scanline descriptions, and performing many of the complex functions listed above. The other program executes on the host computer and provides a command language that humans use to direct the activity of the driver system. The two programs communicate via a communication link. The messages that cross the link include control messages, documents to be printed (both text- and graphics-oriented) and character sets.

Document Preparation Interface In order to make effective use of the printer, document preparation programs must build specially formatted documents. There are two collections of such programs: one collection produces text documents and consists of three generally used programs, while the other collection produces graphics documents and consists of a few generally used programs and a larger group of special purpose programs. The system provides, therefore, a collection of interface operations used to build text- and graphics-oriented documents.

Document Analysis Interface Several programs must interpret documents intended for the scanline printer. One program is provided for debugging documents (or document preparing programs!) and prints documents on ordinary terminals with the control codes decoded into readable form. Another program extracts portions of a text-oriented document for selective printing. A third program displays a page of text on a graphics terminal. A fourth combines several graphics-oriented files to produce a composite graphic.

In order to support these and other programs, the system provides a set of functions that allow the orderly decomposition of documents. These functions are also used internally, in the driver, to decompose the documents for transmission and printing.

Character Set Manipulation One of the primary strengths of the printer system is its ability to use arbitrary character set definitions. The system therefore provides programs that process character set definitions, allowing members of the user community to create and modify character sets on a graphics terminal (with a tablet input device).

In addition, the system maintains a collection of standard character sets with known attributes. Some of these character sets are resident on disk attached to the minicomputer while others must be transmitted from the host before being used by a document.

Character sets are also useful outside the context of the driver. There are programs that permit users to label portions of graphic documents with the character set flexibility normally available to the text-document user. And various ways of displaying the character sets are devised by people who design new character sets. Therefore, a "type" definition of a character set is also exported for general use.

Documentation There is a manual that describes the hardware and software systems that constitute the printer support system. The printer operation is described both for general interest and for the driver program specifications. The driver system and character set

---

*This dissertation is a text-oriented file with embedded graphics-oriented files.

editor are documented from the viewpoint of a user of the system. Examples of many of the standard character sets are displayed to assist the user in character set selection.

There are also some other documents for the system. A list of character sets is displayed in the terminal room. An on-line "help" command prints a short description of the system and a pointer to a documentation file. The driver program also has a help facility and contains text designed to assist the user of the program while it is operating. Another set of character set samples, more complete than the set in the manual, is maintained in a binder near the printer itself.

6.1.2 Environment of Operation of Target System

In general, the details of the environment in which the system operates are not particularly important. However, it can be useful for those who are familiar with various components of the environment to make those aspects more concrete, to more firmly establish the context in which this system operates. Therefore, we present here some of the specific characteristics of the environment in which the printer resides.

The printer itself is a prototype Xerox Graphics Printer. The minicomputer stores scanline descriptions in a buffer shared with the printer; the printer interprets these descriptions to produce a sequence of black and white dots. The black dots are transferred, via a CRT, to a xerographic drum like those used in duplicating equipment. The drum transfers ink in the same pattern to continuous paper that is cut at arbitrary intervals. It produces approximately six eleven-inch pages per minute.

The PDP-10 host computer is a general timesharing facility, one of three PDP-10's at the Carnegie-Mellon Computer Science Department installation. All three PDP-10's use the TOPS-10 monitor system with many local enhancements. Other machines in the installation include C.mmp and Cm* (parallel architecture research computers), PDP-11s operating UNIX, and a large collection of special application PDP-11s. The three PDP-10s and C.mmp are connected to the ARPAnet, and all host systems are available from common front-end terminal interface system.

The connection between the PDP-10 and the PDP-11 that controls the printer is a two wire link, one byte oriented communication link similar to those used by terminals, and a higher speed burst device. Local I/O devices on the PDP-11 include only a DECtape drive, a small disk, and a console terminal. The disk is buffer storage for character sets and documents.

The environment also contains a set of vector graphics terminals. Graphics programs exist for drawing arbitrary pictures, engineering diagrams, and plots that are eventually printed on the scanline printer. The character set editor runs on the graphics terminal as well as ordinary terminals.

There are three document preparation programs of varying degrees of sophistication (Runoff, Pub and Scribe), and a couple of programs to prepare plots from tables of data. Various research projects have special programs to display data, such as speech waveforms or multiprocessor utilization charts, on the printer and/or graphics terminal.

The community of users is quite diverse. They include secretaries, graduate students, faculty members and technicians. Within each of these groups there are incorrigible hackers [Weiz76] as well as people who seldom write a program. The system is continuously subject to recommendations for its improvement (or demise).

6.1.3 Available Implementation Tools

The implementation environment for this system is the same timesharing system that acts as host in the printer system. It has been intensively used in support of programming projects for many years and therefore has a well developed collection of programming tools. While the tools are generally more sophisticated than those available on commercial systems, they incorporate no concepts that have not been within the state of the art for several years.

- Bliss and Sail for PDP-10 implementations. Bliss is a sparse, machine oriented language that produces rather efficient object code [Wulf70]. Sail is a semantically rich language with a large run-time support system [Reis76]. Users of both languages have contributed to program libraries for general use.

- Bliss-11 and Macro-11 for PDP-11 implementations. Bliss-11 is related to Bliss and produces quite efficient object code [Wulf72]. The Macro-11 assemblers are preferred by some programmers [Digi73]. Public tools for Bliss-11 support are available, although most of them are oriented toward the C.mmp/Hydra environment [Levi77]. (Users of the UNIX systems use C exclusively.)

- Two text editors, Lined and Teco[Digi72], are used exclusively as interactive editors; there is no convenient batch-style editor or update program.

- BH data base program[Newc74]. This batch-style program permits the definition. of simple collections of information in a one level structure. While inadequate for many purposes, it accomplishes sort/report-style functions.

- Interactive debugging facilities for PDP-10 programs. Both Sail and Bliss are served by special interactive debuggers, and the operating system provides another.

The driver for the system we are discussing is implemented in Sail (on the PDP-10 side) and Macro-11 (on the PDP-11 side). The driver was designed by one group of people, implemented by person not a member of the design group, and has been maintained recently by yet another person. Various people have contributed the remainder of the system.*

6.2 Selection of the Target System

The representation we have developed needs to be applied to a substantive target system. However, examples for thesis use are notoriously hard to select. The example must be sufficiently small that it can be contained in the thesis, and that the quantity of necessary detail does not swamp the important points of the thesis. But it should not be a "toy" and therefore raise doubts about the relevance of the thesis to "real" systems.

6.2.1 Size of Target System

The size of a system is very difficult to express. We will present some trivial objective

*The present author was not involved in any way in the development or maintenance of the system.

data and some accurate subjective descriptions.

Objective Size The driver program consists of approximately 256,000 characters of program text, about half each in PDP-11 assember language and Sail. This total does not include the text of library routines used by the Sail program but represents only program text concerned with controlling the printer.

The character set editor consists of approximately 140,000 characters of program text, of which 2/5 is PDP-11 assembler language and 3/5 is Sail program. Some of the PDP-11 program is concerned with the graphics display; the rest of the system is character set functionality.

The manual for the system contains fifteen fairly dense pages that describe the command language for the driver. In addition to the actual commands, there are about 40 parameters that the user can modify to control the behavior of the driver system.

Subjective Evaluation of Size The purpose of the system is simple--it provides a facility for printing formatted documents. The conceptual size of this aspect of the system is so small that we will not need to devote more attention to it.

The number of different concepts contained within the system is substantial. Aspects of the system address issues in graphics, communication protocols, networking, permanent object definition (the character sets), text processing, command language interpretation, hardware description, user-visible "system state" and operating system interface. The conceptual size of the implementation is therefore quite large for the program sizes given.

This contradiction, between the simplicity of the purpose of the system and the complexity of the implementation necessary to achieve it, makes the system appealing for use as a demonstrator. It is still too large to handle conveniently, but difficulty in that direction is preferred to applying the techniques to a system that is not capable of exploiting the full range of flexibility.

6.2.2 Reality of Target System

This system is not a "toy" system. It meets the real needs of a diverse user community on a daily basis. It is actively maintained for error repair and system enhancement.

It is clear that the functionality of the system is not mathematically tractable; input/output specifications for the system would not capture the behavior of the system as viewed by its users. For example, the recent changes to the system have been alterations in the character set configuration, adjustments due to hardware alterations, improved robustness of network protocols, and smoothing of the user interface. Therefore, we have not constrained ourselves to a target system that has only formal requirements.

The system is definitely complex. The original implementation required a person-year of work by an expert programmer, plus design contributions by a number of other sophisticated computer scientists. It is also clear to users of the system that the design and implementation are neither adequate nor robust; the original effort, then, was not sufficient to produce a high quality system.

Finally, motivation for multiversion representation is present. Various versions of the system have been implemented to handle different communication links, character set configurations, additional features, and general enhancements. It is easy to extend the dimensions along which members of this system family could develop; for the driver alone, we

could replace the PDP-11 with a Nova, convert from TOPS-10 to TENEX, install a new communication link protocol, and so forth.

6.2.3 Implementation Complexity

The complexity of the printer support system does not result entirely from algorithmic richness. If we look at the problem of constructing this system, we can find several sources of complexity.

- This system is implemented to run on more than one type of hardware; in fact, two parts of it must operate simultaneously on two different computers.
- Parts of the system run on the same host on which they were constructed, while other parts run on a slave machine. The interface sections must be implemented for yet a third host.
- The implementation languages are real rather than imagined. The problem of system structuring is independent of the implementation language, although the representation of components might well be improved if an abstraction-oriented compiler were available.*
- The syntax and semantics of the languages are completely incompatible. The documentation language is incompatible with all of the programming languages, as might be expected.

6.2.4 System Content

The types of objects that constitute the system include a hardware description (the printer behavior), operating system descriptions, user interface descriptions, and a variety of data types, such as character sets, text documents, graphics documents, and links. Objects of the various types exist over long intervals and compatibility is a concern for many users.

The actual objects of the system include values for particular character sets, memory sizes, time limits, and control codes. Protocols, such as the link protocol, document shipping protocol, character set shipping protocol, and the handshake/abort/restart protocol are non-trivial and shared by several subsystems.

6.3 Target System Construction Processors

In order to be able to write the examples in the next section, certain processors must be referenced. This section contains a brief description of each general processor and its parameters. In correspondence with our construction notation, the concrete object are specified within parentheses, while the string parameters are implicit and will be specified in the "with clause of a construction rule.

Processor	Description

*Deferring issues to the programming language is not only practically naive, but assumes that some single language will address all of those issues. Any one language, however, will address only a subset of the interesting issues, and there are several problems that are outside the context of a programming language and will therefore never be solved linguistically.

Sail(source) This is the compiler for the Sail language discussed above. The concrete object provided as its parameter is program text and it produces relocatable object code. Optionally a listing file can be produced as a side-effect. String parameters to the execution of the Sail processor control the generation of debugging code, execution profile code, special listing information, and a variety of minor details.

Bliss(source) This is the compiler for the Bliss language also discussed above. The concrete object provided as its parameter is program text and it produces relocatable object code. Optionally a listing file can be produced as a side-effect. String parameters to the execution of the Bliss processor control the generation of debugging information, the generation of routine timing code, special listing information, optimization strategies, and a variety of minor details.

Link(obj,obj) The Link processor is a linkage editor for relocatable programs such as those produced by Sail or Bliss. The first concrete object parameter is a relocatable program (or possibly a collection of such programs). The second is another collection of relocatable programs that are to be included if the original program refers to symbols contained in them (continued transitively). String parameters to this program control the normal complement of details.

Edit(text,cmds) This processor is a text editor. Unfortunately, no conventional notations exist for describing editing operations. We will therefore use the cumbersome notation of Snobol4, which is at least widely understood. Edit performs text editing operations on the first concrete object parameter using Snobol4 program definitions provided by the second concrete object parameter. The string parameter to Edit specifies which program to execute from that set of definitions. Appendix I contains the definitions of several programs available in the Edit program to all text editing programs.

Expand(text,defs) Some editing operations are more conveniently expressed as macros to be expanded in the source text. The Expand processor uses the Snobol4 program definitions in the second concrete object parameter as macros to be expanded in the first concrete object parameter. Macros in the text will be delimited with the characters "{}" and the delimited text will be considered a Snobol function call. Expand evaluates the innermost macro first, and continues until none remain; therefore macros may produce text that represents another macro call. The programs in Appendix I are also available to all defined macros.

Scribe(text) Scribe is a document preparation program that produces a document from the description provided in its concrete object parameter.

BH(data) BH maintains and processes simple data bases. The concrete object parameter is the data base definition, while string parameters control the input and output processing. In some cases below, the BH processing has been sufficiently simple that it has been replaced by text processing instructions; the primary capabilities of BH that are useful here are sorting of the data base, selection of subsets of the data, and generation of text.

Extract(obj) Extract uses the string parameter to decide what portions of the concrete object parameter to extract. This simple processor is used to represent operations that might be more complex. For example, a program that can parse Yfpl programs might extract a set of routines.

Resolve(obj) Invokes acquire at each place that a resource is required in the concrete object parameter. This program is used to force resources to be acquired within a particular context even though no other processing is needed. Resource resolution is usually accomplished during some other processing step such as compilation.

6.4 Examples from the Target System

The examples that appear below are taken directly from the the scanline printer support system. However, for purposes of explication, some compromises with the form and content of the examples has been necessary. Those compromises are of two forms, simplification and idealization. Portions of the system have been simplified when the necessary descriptive detail obscured the points for which the example was selected, or when several similar aspects are representable by a smaller number of exemplars. In these cases, nothing essential has been eliminated; the actual system merely includes more information of the type presented. Aspects of the system have been idealized in the case that the idealized design reflects an improved design or the idealized design can be systematically transformed into the actual design.

The primary idealization occurs in the assumption that the processors, such as Sail, Scribe, Bliss-11, Macro-11 and Bliss, have been modified to directly interrogate the system construction data base with the *acquire* function. In fact, no processor modifications have been made, and the actual resolution of resources is performed by a preprocessing pass implemented with the general Edit processor. To incorporate the Edit in these examples would merely make the construction rules more complex; in those cases in which a processor would be called upon to resolve a resource, we will eventually have to add the Edit step explicitly.

6.4.1 The Printer Support Software Top-Level System

The printer support system is represented, at the top level, by a single system subsystem that provides all of the resources used by programs and people outside the environment of the printer system itself (see Figure 6-1). It provides resources for the construction of text- and graphics-oriented files and for the manipulation and use of character sets, and for printing documents.

This section describes the outline of the top level system and shows the constituent subsystems. Several of them are elaborated in later sections and therefore their subsystem descriptions are abbreviated here.

Resource	Resource Description
Kset. . .	The resources that contain information about character sets.
Print	A program that operates the printer and produces documents.

```
subsystem PTRSYS
    provides Print*, SelectPages*, RemotePrint*,
        KsetEditor*, KsetList*, KsetType*
        ToffGenerate*, ToffDecode*, GoffGenerate*, GoffDecode*,
        CcGenerate*, CcDecode*, CcPrint*, CcStateSet*, CcDefinition*
    subsystem DRIVER provides Print*, RemotePrint*
        subsystem MASTER provides Print . . . realization . . . end MASTER
        subsystem SLAVE . . . realization . . . end SLAVE
        subsystem PROTOCOL . . . realization . . . end PROTOCOL
        subsystem REMOTE provides RemotePrint . . . realization . . . end REMOTE
        realization . . . end DRIVER
    subsystem KSETS provides KsetEditor*, KsetList*, KsetType* . . . end KSETS
    subsystem TOFF provides ToffGenerate, ToffDecode . . . end TOFF
    subsystem CC provides CcGenerate, CcDecode, CcStateSet, CcPrint,
            CcDefinition . . . end CC
    subsystem GOFF provides GoffGenerate, GoffDecode, GoffBuild . . . end GOFF
    subsystem SELECT provides SelectPages . . . end SELECT
    subsystem SL provides Scanline . . . end SL
    realization
        version Document select DRIVER=Doc, TOFF=Doc, GOFF=Doc,
                KSETS.KD=Doc, KSETS.KE=Doc, SL=Doc
            component Scribe(file(<Printer Manual Source>))
            end Document
        version Executable select DRIVER=Current end Executable
    end PTRSYS
```

Figure 6-1: Top-Level Subsystem

SelectPages	A program that selects pages from a text-oriented file.
RemotePrint	A program that permits users on computing systems not directly attached to the printer to transmit documents across the network and have them printed.
Toff. . .	Facilities for creating and processing text-oriented files.
Goff. . .	Facilities for creating and processing graphics-oriented files.
Cc. . .	Facilities for manipulating the control codes that are present in text-oriented files.

In the subsystem description in Figure 6-1, we have used the abbreviation (introduced in section 4.4.1.3) that appends provided resources to the required resource list. All of the resources provided by the top level subsystem are in fact provided by internal subsystems and "passed through". This structure reflects the designer's impression that those resources are all part of a "printer system" rather than merely the pooled resources of several independent systems. This judgement can be substantiated technically by noting the high degree of cross-connection among the internal subsystems of this system. Users will need to specify the version of the internal subsystem when actually using any resource, as proposed

in section 4.2.2.3. In addition, the top level subsystem prevents dispersal of resources provided by the internal subsystems for use within the system but not intended for use by the public (e.g. KsetXfer).

The DRIVER subsystem contains the programs that actually drive the printer. The MASTER subsystem includes those portions of the system that execute on the host computing system (this subsystem is expanded in detail in sections 6.4.11, 6.4.13, 6.4.14, and 6.4.10). The SLAVE subsystem includes those portions of the system that execute on the dedicated minicomputer. The PROTOCOL subsystem defines the communication protocols between the master and slave programs; these protocols are built on a basic communication protocol provided by a library subsystem.

The KSETS, TOFF, CC, GOFF and SELECT subsystems provide resources used both within this system and by users of this system. These subsystems are described in depth in sections 6.4.6, 6.4.2, 6.4.3, 6.4.4 and 6.4.10 respectively.

The SL subsystem provides the software definition of the printer scanline interpreter, that is, the encoding used to represent sequences of white and blank dots. It is elaborated in section 6.4.5.

What does it mean for the this level of the printer support system to "exist"? In practice, the users identify the printer system with the driver programs and the manual that describes them; therefore, the versions of the top-level system described above include the manual and the running program. The manual must be constructed from the source document <Printer Manual Source> while the driver program is constructed during the construction of the "current" version of the subsystem that provides the "print" resource. (Recall from section 5.3.2 that "building" a version of a system means constructing all components of that version and the components of the specified versions of all internal subsystems.) The other objects that are part of the system are resource definitions, and each of them is constructed as needed.

6.4.2 Text-Oriented File Format

The format of text-oriented files is exploited by the printer driver, the document preparation programs, the file display program, and the page select program. Most programs either generate of process such a file, so there are two complementary views of that format, a generation view and a decoding view.

A text-oriented file is a sequence of units, each of which may be a text string or a control code string. A goal of this definition is to assure users of the printer system that text-oriented files are read and written compatibly, i.e. that all information written as text is processed as text, and that all control information is processed as control information. Each high level version of the text-oriented file format subsystem, then, provides a compatible set of both the generation and decoding resources.

The resource ToffGenerate contains generation operations that extend an output stream with strings of bytes. Rather than associate the particular output destination with these operations, we assume that an output procedure has been provided to the generation routines. Since that output procedure is likely to be common within a program, it is specified globally, not with each function reference. Suppose then that "ToffOutputAppend(byte)" is defined. The operations for generating text-oriented files are the following:

Operation	Operation Description

```
subsystem TOFF
    provides ToffGenerate, ToffDecode
    realization
        version Current
            version Sail
                resources ToffGenerate: file(<Sail Toff Gen Macros>),
                    ToffDecode: file(<Sail Toff Dcd Macros>)
                end Sail
            version Bliss
                resources ToffGenerate: file(<Bliss Toff Gen Macros>),
                end Bliss
            version Macro11
                resources ToffDecode: file(<Macro11 Decode Macros>)
                end Macro11
            end Current
    end TOFF
```

Figure 6-2: Text-Oriented File Format Subsystem

ToffText(string) Emits the characters in the string as a text unit.

ToffCtls(string) Emits a the string as a control code unit.

The decoding operations are used quite differently, since they are invoked as a result of a computation (input from the document). There is one operation to extract the next unit of information from the document (corresponding to the units emitted by the functions above) and an operation to process the contents of the unit discriminated by its type. Again, in order to separate the decoding from the input source, the extraction operation operates on a string.

Operation	Operation Description
ToffNext(string,<succ>,<fail>)	This operation removes the next unit from the string. If it succeeds, it makes that unit available to the <succ> action. If it is unable to extract a unit from the string provided, it executes the <fail> action. If the behavior of the processing program is straightforward, this operation could be inserted into a loop with the <succ> action performing the processing and the <fail> action appending more data from the input source. For Sail programs, this operation is a Sail statement.
ToffPartition(<action1>,<action2>)	This operation performs one of the specified actions based on the type of the current unit. Each action is a pair, the first element of which is either "text" or "ctls" and the second is the corresponding program text. The operation is valid only within the context of ToffNext (presumably as part of the <succ> action) and, for Sail programs, is a Sail statement.

These resources will be used in a complete construction example in section 6.4.10. Therefore, the Sail version of the resources, that is, the contents of files called <Sail Toff Gen Macros> and <Sail Toff Dcd Macros> are reproduced in Appendix II.

This system has a short hierarchical version structure. The version "Current" represents a compatible set of resources, i.e. the generate facility produces file that can be read by the decode facility. (No alternative to "Current" is provided in the subsystem description above.) Within the "Current" version are versions corresponding to each of the programming languages. A specific leaf version is indicated with a path (e.g. Current.Sail) as described in section 4.2.2.3.

Because ToffGenerate is used by both Sail and Bliss programs, and ToffDecode is used by both Sail and Macro11 programs, each resource appears is two language versions. The two missing versions (Bliss decode and Macro11 generate) have never been implemented and never will be; we predicted this lack of orthogonality in the implementation of orthogonal variability in section 3.2.1.

6.4.3 Text File Format Control Codes

As mentioned in section 6.4.2, there are a number of control codes that can be embedded in a text-oriented file. Some of these control codes set parameters in the driver system to control the margins, spacing, justification, and page size. Others invoke special facilities of the driver such as subscripting, underlining, and overstriking. Still others embed a graphics-oriented file or execute a driver command, e.g. transfer a character set from the host to the slave. A subset of the control codes is given in Figure 6-3 and the subsystem is shown in Figure 6-4.

Resource	Resource Description
CcDefinition	The original control code definition table.
CcGenerate	For each control code there is an operation that creates the string representing the control code with its parameter. For Sail programs, these operations are string expressions. For example, CcEOL is a string containing an end-of-line control code, CcTM(100) is a string containing a control code to set the top margin to 100, and CcUND("important text") contains a control code to underline a string.
CcDecode	This resource consists of two operations, CcNext and CcPartition, that are used to decode strings of control codes.

Operation	Operation Description
CcNext(string,<succ>,<term>,<fail>)	This operation removes the first control code from the string of control codes provided and, if it succeeds, makes the control code available to the <succ> action. If the string is empty, the <term> action is executed, and if the string does not contain a valid control code, the <fail> action is executed. For Sail programs, this operation is a statement, and the actions to be performed are statements. This operation would be an appropriate action to be associated with a control code string unit in

Control Code	Internal Value	Parameter Type	Initial Value	Effect Description
VS	1	Integer*2	7	set vertical spacing
LM	2	Integer*2	200	set left margin
TM	3	Integer*2	200	set top margin
BM	4	Integer*2	200	set bottom margin
LIN	5	Integer*2	55	set number of lines per page
LA	6	Integer*1	4	load A character set from disk
LB	7	Integer*1	0	load B character set from disk
UA	8			use A character set
UB	9			use B character set
JW	10	Integer*2	0	set justify width
PAD	11	Integer*2	0	set maximum padding
SP	12	Integer*1		variable-length blank
EOL	13			end of line
EOP	14			end of page
TAB	15	Integer*2		tab to raster position
QU	16	String		quotes the string
OVR	17	String		overstrike the string
SUP	18	String		superscript a string
SUB	19	String		subscript a string
DCP	20	String		decapitate a string
UND	21	String		underline a string
SL	22	Integer*2	2200	set number of scanlines per page
BAK	23	Integer*2		backspace
HD	24	String	Null	set heading line skeleton
HN	25	Integer*2	1	set heading page number
BR	26			break at end of page
EOF	27			end of file
CMD	28	String		embedded driver command
GR	29	String		embedded graphics file

Figure 6-3: Control Code Table

```
subsystem CC
    provides CcGenerate, CcDecode, CcStateSet, CcPrint, CcDefinition
        realization
        concrete object SailCcEdit=file(<Sail CC Edits>),
            BlissCcEdit=file(<Bliss CC Edits>),
            Macro11CcEdit=file(<Macro11 CC Edits>)

        version Current
            concrete object Def=file(<Control Code Definition>)
            version Sail
                resources CcGenerate: Edit(Def,SailCcEdit) with "CcGenerateSail"
                    CcDecode: Edit(Def,SailCcEdit) with "CcDecodeSail"
                    CcPrint: Edit(Def,SailCcEdit) with "CcPrintSail"
                    CcStateSet: Edit(Def,SailCcEdit) with "CcStateSetSail"
                end Sail

            version Bliss
                resources CcGenerate: Edit(Def,BlissCcEdit) with "CcGenerateBliss"
                    CcDecode: Edit(Def,BlissCcEdit) with "CcDecodeBliss"
                    CcPrint: Edit(Def,BlissCcEdit) with "CcPrintBliss"
                    CcStateSet: Edit(Def,BlissCcEdit) with "CcStateSetBliss"
                end Bliss

            version Macro11
                resources
                    CcDecode: Edit(Def,Macro11CcEdit) with "CcDecodeMacro11"
                    CcStateSet: Edit(Def,Macro11CcEdit) with "CcStateSetMacro11"
                end Macro11
            version Doc resources CcDefinition: Def end Doc
            end Current

        version Development
            concrete object Def=file(<New Control Code Definition>)
            . . . end Development
    end CC
```

Figure 6-4: Control Code Subsystem

ToffPartition.

CcPartition(ctl,<actions>) Each of the actions is a pair associating a control code with an action. This operation executes the action associated with the current control code. This operation is valid only within the context of CcNext (presumably as the <succ> action) and, for Sail programs, is a statement. Occurrences of codes delimited by "⊂⊃" in the actions are replaced by various values: ⊂p⊃ is replaced by the value of the corresponding parameter and ⊂g⊃ is replaced by an invocation of the corresponding operation from CcGenerate. A special control code, "*", indicates the default action for all control codes not explicitly

associated with an action.

CcStateSet Control codes in the table with an initial value attribute describe the system state, and values of those codes are retained until they are reset. It is useful in some programs to treat the members of the state set systematically. This resource provides the CcStateSet(template,separator) operation that results in a list of the templates instantiated for each member of the state set. Occurances of codes delimited by "⊂⊃" in the template are replaced by various values: ⊂c⊃ is replaced by the code name, ⊂t⊃ is replaced by the type of the parameter, ⊂i⊃ is replaced by the initial value of the parameter, ⊂v⊃ is replaced by the internal value of the control code, ⊂d⊃ is replaced by the description, ⊂g<⊃ is replaced by the left portion of the corresponding CcGenerate operation for the code and ⊂g>⊃ is replaced by the right portion of that operation (bracketing the parameter value). For Sail programs, the template might be a declaration or statement with the separator ";", or the template might be an expression with the separator ",".

CcPrint The control codes have a conventional print format produced by the operations in this resource. The operation CcPrint() is valid within a CcNext context (such as the <succ> action of CcNext). For Sail programs, this operation is a string expression with a value such as "UB" or "BAK=193".

These resources will also be used in the example in section 6.4.10. File <Control Code Definitions> is given in Figure 6-3; each code is stored as a line in the file and the columns are separated by tab characters. File <Sail CC Edits> and the intermediate files that contain the Sail versions of the resources are reproduced in Appendix III.

The alternative "Development" version of the CC subsystem shares the editing commands with the "Current" version but has its own control code definition file. If the new version differed not in the actual codes but in the way they were represented in programs, the control code file would be defined globally and each version would have a set of editing commands.

Each resource is a set of macros that will expand into the appropriate programming language text for the operation. Since both editing commands and macros are being represented here as Snobol4 programs, these editing commands are Snobol4 programs that produce other Snobol4 programs that produce Sail (or Bliss or Macro11) program segments.

The intermediate step is used because the control code definitions are used to create a variable number of operations (one for each code in the defining table) that are themselves parameterized; e.g. the "superscript" operation takes a string parameter.

6.4.4 Graphics-Oriented File Format

Graphics-oriented files are sequences of scanline descriptions, blanks spaces and paper cuts. Scanline descriptions are basically the same sequences of 8-bit bytes that are interpreted by the printer hardware as described in section 6.4.5. Programmers usually view graphics-oriented files as two-dimensional bit matrices. To support this view, there need to be operations to draw vectors and character strings in such planar arrays.

```
subsystem GOFF
    provides GoffGenerate, GoffDecode, GoffBuild
    requires SLDefinition, KsetType, KsetList, SailProgramEdits . . .
    external SL, KSET, SAIL
    realization
        version Sail select SL=ref, KSET.KT=Sail, KSET.KD=SailAll . . .
            concrete object Source=file(<Goff Gen Source>),
                    GenSource=Extract(Source) with 1,
                    DcdSource=Extract(Source) with 2,
                    BldSource=Extract(Source) with 3,
                    Edits=acquire(SailProgramEdits)
            resources GoffGenerate: Edit(GenSource,Edits) with "Headers"
                GoffDecode: Edit(DcdSource,Edits) with "Headers"
                GoffBuild: Edit(BldSource,Edits) with "Headers"
            deferred GoffGenerate: Sail(GenSource)
                GoffDecode: Sail(DcdSource)
                GoffBuild: Sail(BldSource)
            end Sail
    end GOFF
```

Figure 6-5: Graphics-Oriented File Format Subsystem

Resource	Resource Description
GoffDecode	Facilities for reading graphics-oriented files and processing the successive elements.

Operation	Operation Description
GoffNext(buffer,<succ>,<fail>)	This operation removes the next unit (either a scanline, paper cut, or blank space) from the buffer. If it succeeds in extracting a unit, it makes it available to the <succ> action, otherwise it executes the <fail> action.
GoffPartition(<actions>)	Each of the actions is a pair that associates the keys SL, BL, CUT and EOF with corresponding actions. A likely action for the SL key would be the ScanlineImage operation from the SLDecode resource in the SL subsystem.

Resource	Resource Description
GoffGenerate	These operations insert units into an output buffer. As in the text-oriented file format case, we assume that the operation GoffOutputAppend(byte) is defined where these operations are used.

Operation	Operation Description
GoffScanline(bitvector)	Emits a scanline description of the bitvector.
GoffBlankSpace(n)	Emits n scanlines of blank space.

	GoffCut	Emits a paper cut.
	GoffEof	Emits an end of file indicator.

GoffBuild This resource contains the two-dimensional bit array operations that are useful for constructing graphics-oriented files when a single scanline at a time is inappropriate.

Operation	Operation Description
GoffInit(x,y)	Establishes a buffer x bits by y bits.
GoffClear	Sets the buffer to zeros.
GoffKset(ksetid)	Loads the indicated character set.
GoffBit(i,j)	Sets the (i,j) bit to one.
GoffVector(i1,j1,i2,j2)	Draws a vector from (i1,j1) to (i2,j2).
GoffChar(i,j,string)	Draws a character string starting at (i,j).
GoffWrite()	Writes the buffer out, collapsing blank space.

The set of operations above is an indication of the types of operations available for producing graphics-oriented files; in fact there are many more, such as operations to draw character strings upsidedown and sideways. The GoffBuild resources are used by the program discussed in section 6.4.8 while the *GoffDecode* and *GoffGenerate* resources are used by a program that merges several graphic-oriented files into a composite file. Of course, the GoffDecode operation is also used in the driver, as discussed in section 6.4.11. The relationship between GOFF and SL is hierarchical. GOFF uses the resources of SL to convert from bitvectors to scanline descriptions. GOFF does not attempt to interpret a scanline definitions, just as TOFF does not interpret control code sequences.

6.4.5 Printer Scanline Interpreter

The printer produces black dots on paper, a scanline at a time, from descriptions provided to it by the dedicated minicomputer. The scanline interpreter is capable of generating these bit patterns from a direct bit vector representation or from run codes, which are alternating counts of black and white dots. The finite state machine of Figure 6-6 describes this scanline interpreter. Two abbreviations are used in that diagram. First, bounded counters are named and may be tested on the input side of a transition, and updated on the output side of a transition (boundary violations are errors). Second, the input character is named "b", may be tested on the input side of a transition, may be used as an integer (for updating counters) or as a bit sequence in the output side of a transition. The transition label format is "(test)→output(actions)".

A text representation of the FSM above given in Figure 6-7, could be stored in a text file.

The finite state machine definition is a complete description of the legal sequences of bytes and the resulting scanlines. The operations that are performed on scanlines, however, may or may not be easily derived from this representation. One operation, a program to provide a

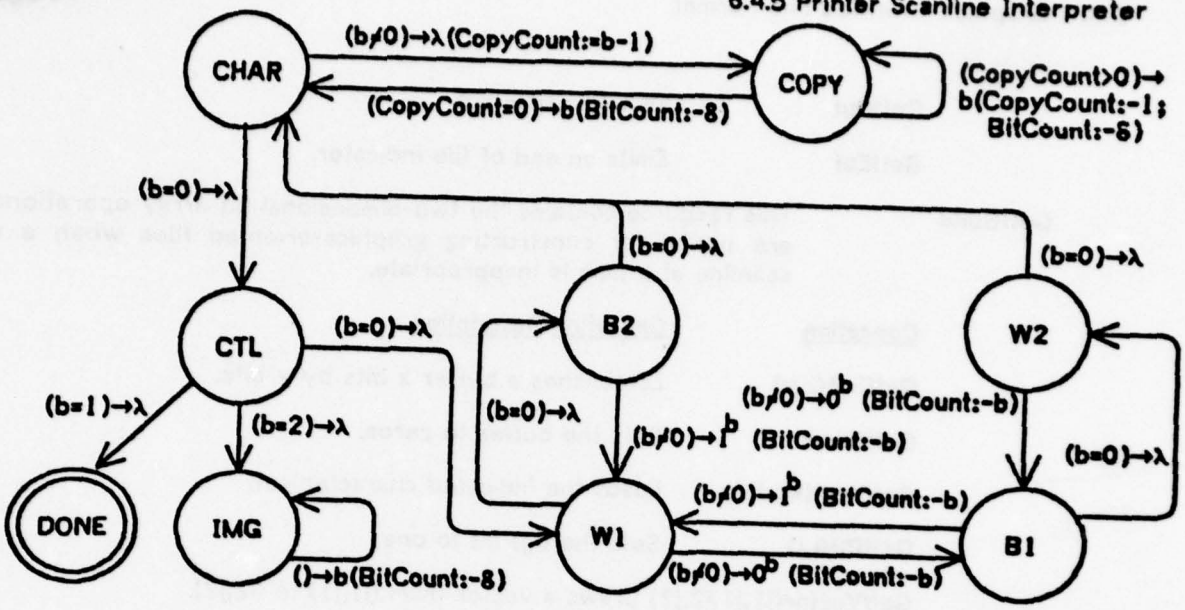


Figure 6-6: Scanline Definition

Old State	New State	Test	Output	Actions
(initialize)	CHAR	λ	λ	BitCnt:-1700
CHAR	COPY	b≠0	λ	CopyCnt←b-1
"	CTL	b=0	λ	
COPY	COPY	CopyCnt>0	b	CopyCnt:-1; BitCnt:-8
"	CHAR	CopyCnt=0	b	BitCnt:-8
CTL	W1	b=0	λ	
"	DONE*	b=1	λ	
"	IMG	b=2	λ	
IMG	IMG	TRUE	b	BitCnt:-8
W1	B2	b=0	λ	
"	B1	b≠0	0^b	BitCnt:-8
B1	W1	b≠0	1^b	BitCnt:-8
"	W2	b=0	λ	
W2	B1	b≠0	0^b	BitCnt:-8
"	CHAR	b=0	λ	
B2	CHAR	b=0	λ	
"	W1	b≠0	1^b	BitCnt:-8

Figure 6-7: Scanline Definition Table

bit vector representation of a scanline, can be produced easily from the description, using any suitable representation of finite state machines with a special "hook" for processing output descriptions. However, it is not trivial to produce a program that will produce a compressed representation of a scanline (in which long streams of zeros or ones are replaced by run codes) nor to produce a set of verification conditions that will guarantee that the compressor will produce equivalent scanlines.

Resource	Resource Description
SLGenerate	This resource contains a single operation that converts a bitvector *(in a language dependent representation) to a scanline* description, which is a sequence of 8-bit bytes. It optionally exercises the SLCompress operation on the result.
SLDecode	This resource contains an operation that converts a scanline into a bitvector.
SLCompress	For those programs that deal in uninterpreted scanlines, this resource contains an operation to optimize the space occupied by the description of a given set of bits.

The Yfpl program in Figure 6-9 is the output of a hypothetical FsmYfplGen processor that is capable of generating programs from the tabular representation of FSM's used above. *Since this program is constructed directly from the definition of the FSM,* a modification to the FSM definition would be automatically propagated to all programs that used the Yfpl version

```
subsystem SL
    provides SLGenerate, SLCompress, SLDecode, SLDefinition*
    subsystem SL'
        provides SLDefinition
        realization
            concrete object Def=file(<Scanline Definition>)
            version Fsm resources Def end Fsm
            version Ref resources "Comment Scanline Use" components Def end Ref
            version Yfpl resources FsmYfplGen(Def) end Yfpl
        end SL'
    realization
        version Sail select SL'=Ref
            concrete object DefTag=acquire(SLDefinition)
            resources SLGenerate: Concat(DefTag,file(<Scanline Gen Functions>)),
                SLCompress: Concat(Deftag,file(<Scanline Cmpr Functions>)),
                SLDecode: Concat(DefTag,file(<Scanline Dcd Functions>))
            end Sail
    end SL
```

Figure 6-8: Scanline Interpreter Subsystem

```
var BitCount, CopyCount:integer,
   state:{CHAR, CTL, COPY, W1, W2, B1, B2, IMG, DONE}
procedure Decr(ctr:ref integer, n:integer);
   if ctr≥n then ctr←ctr-n else ERROR()
   end Decr
procedure ScanlineInit(); state←CHAR; BitCount←1700 end ScanlineInit
procedure ScanlineNext(b:byte);
   case state of
      CHAR: if b=0 then state←CTL else CopyCount←b-1; state←COPY fi
      COPY: if CopyCount=0 then output(b); Decr(BitCount,8); state←CHAR
            else output(b); Decr(CopyCount,1); Decr(BitCount,8)
      CTL:  if b=0 then state←W1
            elseif b=1 then state←DONE
            elseif b=2 then state←IMG
            else ERROR() fi
      DONE: ERROR()
      IMG:  output(b); Decr(BitCount,8)
      W1:   if b=0 then state←B2
            else for j←1 thru b do output(0); Decr(Bitcount,b); state←B1 od fi
      B1:   if b=0 then state←W2
            else for j←1 thru b do output(1); Decr(BitCount,b); state←W1 od fi
      W2:   if b=0 then state←CHAR
            else for j←1 thru b do output(0); Decr(Bitcount,b); state←B1 od fi
      B2:   if b=0 then state←CHAR
            else for j←1 thru b do output(1); Decr(BitCount,b); state←W1 od fi
      end case
   end ScanlineNext
```

Figure 6-9: Finite State Machine Generated Program

of the scanline definition.

Other programs are unable to use a mechanically produced version of the scanline definition. For example, the program that performs the SLCompress operation translates one scanline description into an equivalent but shorter scanline description. Handcoding this procedure is the appropriate use of state of the art techniques. However, in order to record the use of the information contained in the SLDefinition resource, the program *acquires* a version of the resource that is a comment. By attaching appropriate policies to each of the pieces of the subsystem, modification of the definition file will send a message to the Ref version (of which the definition file is a component) that can then send messages to each of its users. The handcoded compress routine, upon receiving that message, can prevent further use of itself until a programmer has re-established the correspondence between the program and the definition, and can send messages to each of its users indicating that the program is currently incompatible with the current definition.

6.4.6 Character Set Definition and Directory

The character sets that are used to produce the text scanlines are used in several places within the system as well as by other, related systems. A character set has a fixed height

and baseline; the height is the number of scanlines in each grid while the baseline is the position in the grid that is used to align character sets. Each of the 128 grids in the character set has its own width and increment; the width is the actual number of scan positions in the grid, while the increment is the amount by which the scan pointer should be moved (thereby allowing overlap of character grids and avoiding storage of blank grid columns).

Resource	Resource Description
KsetType	A type definition for the manipulation of a single character set
KsetList	A list of the available character sets and their characteristics
KsetXfer	A mechanism for transferring the representation of a character set from one medium to another.
KsetEditor	A program to create or edit character set definitions.

The original definition of the character set directory is the file that is given the tag "Def" above. Figure 6-11 represents the contents of that file; only a small number of character sets are listed here, since the actual character set directory has over 100 entries.

Tag	Number	Height	Base	Width	Std	Description
CLAR35	53	35	29	Var	No	Clarendon Roman
FIX25	4	25	20	16	Yes	News Gothic Fixed Width
BDR40	6	40	30	Var	No	Bodoni Roman
LPT128	83	28	22	16	Yes	Line Printer Simulator
NGB25	11	25	20	Var	Yes	News Gothic Bold
NGB30	19	30	24	Var	No	News Gothic Bold
NGB40	127	40	32	Var	No	News Gothic Bold
NGI20	117	20	16	Var	No	News Gothic Italics
NGI25	1	25	20	Var	Yes	News Gothic Italics
NGR13	22	13	11	Var	No	News Gothic Roman
NGR20	15	20	16	Var	Yes	News Gothic Roman
NGR25	7	25	20	Var	Yes	News Gothic Roman
NGR30	23	30	24	Var	Yes	News Gothic Roman
NGR36	124	36	29	Var	No	News Gothic Roman
NGR40	8	40	32	Var	Yes	News Gothic Roman
APL25	137	25	20	18	Yes	APL Type Ball Simulator

Figure 6-11: Character Set Directory

The Sail versions of the character set directory are parallel vectors with corresponding elements in each row. In another language, a vector of records might be as convenient. The documentation version of the character set directory contains, in addition to the displayed data, the control characters that indicate to Scribe that the data is to be aligned in columns.

The KsetType definition is shown above as a set of four pages in a file that are maintained in parallel. Figure 6-12 shows a Yfpl program skeleton that represents the operations

```
subsystem KSET provides KsetEditor*, KsetList*, KsetType*, KsetXfer*

    subsystem KE provides KsetEditor
        requires KsetEditorDriver, KsetEditorMonitor
        subsystem KED provides KsetEditorDriver
            requires KsetType, KsetXfer, KsetList, SailExtensions
            external KT, KX, KD, SAIL
            realization
                version One select KT=Sail, KX=Sail, KD=SailAll, SAIL=Std
                    component Link(Sail(file(<Kset Editor Driver Source>)))
                    end One
            end KED

        subsystem KEM provides KsetEditorMonitor
            requires KsetType, GraphicsMonitor
            external KT, GRAPHICS
            realization
                version One select KT=Bliss11, GRAPHICS=Bliss11
                    component Link11(Bliss11(file(<Kset Editor Monitor Source>)))
                    end One
            end KEM
    realization
        version One select KEM=One, KED=One end One
        end KE

    subsystem KT provides KsetType
        realization
            concrete object Types=file(<Kset Type Definition>)
            version Sail resources Extract(Types) with 1 end Sail
            version Bliss11 resources Extract(Types) with 2 end Bliss11
            version Bliss resources Extract(Types) with 3 end Bliss
            version Macro11 resources Extract(Types) with 4 end Macro11
        end KT

    subsystem KX provides KsetXfer
        realization . . . end KX
    subsystem KD provides KsetList
        realization
            concrete object Def=file(<Kset Directory Table>),
                Edits=file(<Kset Directory Edits>),
                Std=BH(Def) with "select:'Std=Yes'"
            version SailAll resources Edit(Def,Edits) with "SailVector" end SailAll
            version SailStd resources Edit(Std,Edits) with "SailVector" end SailStd
            version Doc resources Edit(Def,Edits) with "ScribeTable"  end Doc
        end KD

    realization
    end KSETS
```

Figure 6-10: Character Set Definition Subsystem

contained in each of those pages in a form suitable for each language.

```
type Kset
  operation KsetName returns string
      KsetId returns integer
      KsetHeight returns integer
      KsetBaseline returns integer
      KsetWidth(integer c) returns integer
      KsetIncrement(integer c) returns integer
      KsetRow(integer c,i) returns bitvector
      KsetSetRow(integer c,i; bitvector b)
      KsetSetWidth(integer c,w)
      KsetSetIncrement(integer c,i)
      KsetSetHeight(integer h)
  end Kset
```

Figure 6-12: Definition of Type *Kset*

6.4.7 Scribe Document Preparation Program

The community served by the scanline printer generates a large number of theses, papers, books, reports, articles and letters. Scribe has become the program of choice for this purpose; it generates documents for a variety of devices including the scanline printer.* It produces documents for the printer that are not only properly formatted into chapters, sections, and paragraphs, but have also been justified according to the specifications of the actual character sets that will be used. Therefore, it uses the text-oriented file format resource, the directory of character sets, and the character set type definition (as well as resources provided by other subsystems).

```
subsystem SCRIBE
    provides report, letter, thesis, paper, article
    requires ToffGenerate, KsetList, KsetType . . .
    external PTRSYS . . .
    realization
        version Current
            select PTRSYS.TOFF=Bliss, PTRSYS.KD=Bliss.All, PTRSYS.KT=Bliss
            component LINK(Bliss(file(<Scribe Source>)))
            end Current
        version Manual select PTR.KD=mss
            component Scribe(file(<Scribe Manual Source>))
            end Manual
end SCRIBE
```

Figure 6-13: Scribe Subsystem

---

*This thesis was produced with Scribe.

6.4.8 Spacs Picture Drawing System

The Spacs Picture Drawing System provides a general picture drawing facility for the graphics terminals available in the installation*. One of the programs in that system converts a Spacs picture, which is comprised of characters and vectors, into a graphics-oriented file for the printer. This program uses the high level graphics-oriented file format resources; they in turn utilize the character set type definition, low level graphics-oriented resource and the printer scanline definition (see section 6.4.4).

```
subsystem SPACS
    provides PictureEditor, PictureConverter
    require PictureEditor, PictureConverter
    subsystem PE provides PictureEditor . . . end PE
    subsystem PC
        provides PictureConverter
        requires GoffGenerate . . .
        external PTR
        realization
            version Current select PTR.GOFF=Sail
                component LINK(Sail(file(<Picture Convert Source>)))
                end Current
        end PC
    realization
        version Manual
            component Scribe(Scribe(file(<Spacs Manual Source>))
            end Manual
        version Current select PE=Current end Current
    end SPACS
```

Figure 6-14: Picture Editor Subsystem

6.4.9 Document Typer

Text-oriented files are not suitable for printing on standard terminals. The Document Typer program prints a text-oriented file on a terminal in one of two forms, either eliding the control codes or printing the control codes in a readable form. In order to perform its function it must not only decode the file but be able to print representations of the control information.

6.4.10 Select Pages Program

The Select program extracts pages from text-oriented files. It is available as a program and as a function within the driver. It is the only program that both reads and writes text-oriented files. In order to have the extracted pages function as an independent text-oriented file, the embedded user command must be retained (even though they are in pages excluded from the new file) and various state variables must be maintained, such as the

*The diagrams in this thesis were produced with the Spacs Picture Drawing System.

```
subsystem TYPER
    provides TyperProgram
    requires ToffDecode, CcDecode, CcPrint
    external TOFF, CC
    realization
        version current select TOFF=Current.Sail, CC=Current.Sail
            components Link(Sail(<Document Typer Source>))
            end current
    end TYPER
```

Figure 6-15: Document Typer Subsystem

current character sets and margins.

```
subsystem SELECT
    provides SelectPages
    requires ToffDecode, ToffGenerate, CcDecode, CcGenerate, CcStateSet,
            SailProgramEdits, FileUtilities, StringUtilities
    external TOFF, CC, SFU, SSU, SAIL
    realization select SAIL=lib, SSU=lib, SFU=lib
        concrete object Source=file(<Select Routine Source>)
        version Executable select TOFF=Sail, CC=Current.Sail
            concrete object Bodies=Sail(Expand(Source,
                    acquire(ToffGenerate),acquire(ToffDecode),
                    acquire(CcGenerate),acquire(CcDecode),acquire(CcStateSet))),
                SelectHeaders=Edit(Source,acquire(SailProgramEdits) with "Headers"
            version Subroutine
                resources SelectHeaders
                deferred Bodies
                end Subroutine
            version Program
                concrete object SelectMain=Concat("begin ", SelectHeaders,
                        file(<Select Main Program>), " end")
                components Link(Sail(SelectMain),Bodies)
                end Program
            end Executable
        version Document components Scribe(<Select Manual Source>) end Document
    end SELECT
```

Figure 6-16: Select Pages Subsystem

The construction details of this example have been carried out in full. The contents of the files <Select Routine Source> is reproduced in Appendix IV.1 with the I/O routines deleted. The TOFF and CC operations are implemented as macros, and are therefore surrounded by braces. The Sail versions of the macros from TOFF and CC are in Appendices II and III, respectively. The result of the expansion of the Select source with those macros is presented in Appendix IV.2.

6.4.11 Driver--Master Side

The printer driver is the central facility of the software support system. Users issue commands to the host program, which then sends messages to the slave program containing text- and graphics-oriented files, character sets, and special protocol messages (start of transmission, abort, etc.). A variety of operating system facilities are used to control the interactions of two or more users who are attempting to print documents at the same time.

The master side of the driver program is divided into subsystems that provide operating system facilities, the command language interpreter, and the command language executors. A spooling version of the driver does not actually interact with the printer but queues commands for the real driver to execute at a later time. Two of the subsystems will be further elaborated in later sections.

```
subsystem MASTER provides Print
    requires SailExtensions, Protocol, CommandLanguage
    external SAIL, PROTOCOL
    subsystem OPSYS provides AbortTrap*, AdjustPriority*, Delay*, Alloc*, Ppn*
        external PPN, AT
        subsystem PRIORITY provides AdjustPriority . . . end PRIORITY
        subsystem ALLOC provides Alloc . . . end ALLOC
        subsystem DELAY provides Delay
            requires Tingle external TINGLE . . . end DELAY
        realization . . . end OPSYS
    subsystem CL provides CommandLanguage, Commands, Variables
        requires HelpCmdEx, ShipCmdEx, SelectCmdEx, KickCmdEx
        realization . . . end CL
    subsystem HELP provides HelpCmdEx
        requires Commands, Variables external CL
        realization . . . end HELP
    subsystem SHIP provides ShipCmdEx
        realization . . . end SHIP
    subsystem SELECTOR provides SelectCmdEx
        requires SelectPages external SELECT
        realization select SELECT=Executable.Subroutine
            version Only
                resources Concat( "begin ", acquire(SelectPages),
                        file(<Selector Program Source>), " end")
                end Only
        end Selector
    realization
        version Direct select CL=Direct, OPSYS=Only, SAIL=lib
            concrete object DirectMain=Sail(file(<Direct Master Source>))
            component Link(DirectMain,deferred(DirectMain))
            end Direct
        version Spool select CL=Spool, SAIL=lib
            concrete object SpoolMain=Sail(file(<Spool Master Source>))
            component Link(SpoolMain,deferred(SpoolMain))
            end Spool
    end MASTER
```

Figure 6-17: Driver--Master Side Subsystem

Several levels of resource indirection occur in this subsystem as provided for in section 4.4.1.3. First, MASTER provides as resource eventually provided again by PTRSYS (see section 6.4.1). OPSYS provides a set of resources some of which are available in the library while others are locally implemented.

An exaggerated need for access to subsystems defined at relatively high block levels (see section 4.2.2.3) is illustrated by the DELAY subsystem which requires Tingle[Nels77]. This resource is require by DELAY but that need does not follow from the structure of the enclosing systems OPSYS, MASTER, DRIVER or PTRSYS. Therefore, DELAY must be able to directly name library subsystems.

A circular interconnection is necessary in this subsystem, as predicted in section 4.1 and discussed in section 7.1.2.3. The CommandLanguage resource defines the syntax of the commands and requires command executors from other subsystems. One of those command executors, the HelpCmdEx resource, requires descriptive text about the commands to display to the user. This natural cross-connection causes no problem and is easily managed.

6.4.12 Remote Print Program

Some printer users work on systems connected to the printer system only by way of a network. This program transfers the documents to be printed to the host system, runs the driver program and prints the documents. This program interprets the responses of the driver program to determine when the document has been printed whereupon it deletes the transferred copy and returns to the human user.

Maintenance of this program is difficult to automate. The particular knowledge it uses of the command language is small and not likely to change. An appropriate use of the policy mechanism in the construction database (see section 5.3.4) would be to notify the human maintainer of this program each time the command language changes. The human then determines whether any reconstruction of this subsystem is necessary.

```
subsystem REMOTE provides RemotePrint
    requires CommandLanguage, NetworkProtocol external CL, NETWORK
    realization select CL=Ref, NETWORK=Telnet
        version Only
            component Link(Sail(file(<Remote Print Source>)
            end Only
    end REMOTE
```

Figure 6-18: Remote Print Subsystem

6.4.13 Driver Command Language

This subsystem defines the command language for the driver system discussed in section 6.4.11. The CommandLanguage resource is program text to recognize commands and invoke the appropriate command executors. The Commands and Variables resources provide text for the Help command executor.

There are two programs that cause document printing, one that directly drives the printer and another that queue requests for delayed printing. The command languages for these two

```
subsystem CL provides CommandLanguage, Commands, Variables
    requires HelpCmdEx, ShipCmdEx, SelectCmdEx, KickCmdEx
    external HELP, SHIP, SELECTOR, KICK
    realization
    concrete object CmdDef=file(<Command Definitions>),
                    VarDef=file(<Variable Definitions>)
        version Ref
            resources "Comment Driver Command Language"
            components CmdDef, VarDef
            end Ref
        version Direct
            select SELECTOR=Only, HELP=Direct, KICK=InLine, SHIP=InLine
            concrete object DirectCmds=BH(CmdDef) with "select:Subset=Direct",
                DirectVars=BH(VarDef) with "select:Subset=Direct"
            resources Commands: DirectCmds, Variables: DirectVars,
                CommandLanguage: Concat( "begin ",
                    Edit(DirectCmds,Edits) with "Commands", ";",
                    Edit(DirectVars,Edits) with "Variables", ";",
                    acquire(ShipCmdEx), ";", acquire(HelpCmdEx), ";",
                    acquire(SelectCmdEx), ";", acquire(KickCmdEx), ";",
                    file(<Direct Driver Source Program>), " end")
            end Direct
        version Spool select HELP=Spool, SHIP=Spool
            subsystem Spool1 provides KsetList* external KSETS
                realization select KSETS.KD=SailAll end Spool1

            concrete object SpoolCmds=BH(CmdDef) with "select:Subset=Spool",
                SpoolVars=BH(VarDef) with "select:Subset=Spool"
            resources Commands: SpoolCmds, Variables: SpoolVars,
                CommandLanguage: Concat("begin ",
                    Edit(SpoolCmds,Edits) with "Commands", ";",
                    Edit(SpoolVars,Edits) with "Variables", ";",
                    acquire(HelpCmdEx), ";", acquire (ShipCmdEx), ";",
                    acquire(KsetList), ";",
                    file(<Spool Driver Source Program>), " end")
            end Spool
    end CL
```

Figure 6-19: Command Language Subsystem

programs overlap extensively but each has unique features absent from the other; for example, the direct driver permits the user to intervene in the slave environment while the spooler allows queue parameters to be specified. Hence, the CL resources occur in two versions and therefore the Help command executor will exist in two versions (see section 6.4.14). The commands and variables descriptions are extracted from a combined definition.

For reasons without technical validity, the direct driver requires the user to explicitly transfer some character sets to the slave before printing documents that use them. The spooler automatically inserts those commands if they are needed. In order to accomplish this, it must know which character sets are already present, information that is contained in the character set directory resource. As provided in section 4.4.1.4, a new subsystem is

introduced for the spooler version; in this case it merely indicates the library subsystem and version to be used for the newly introduced resource.

The SELECTOR subsystem provides the Select command executor. It is the same function as the SelectPages resource described in section 6.4.10, with the following modification: the *select command automatically provides a temporary file* for the selected pages and then invokes the Print command executor.

The SELECTOR subsystem also illustrates the flow of a deferred object. The "acquire(SelectPages)" phrase in the construction of the SelectCmdEx resource object causes the deferred object associated with SelectPages (namely, "Bodies" in the SELECT subsystem) to be attached to the SelectCmdEx object. It will be attached to the result of the Concat processor in CL, and again to the DirectMain object when the Sail compiler uses the CommandLanguage resource. The Link processor takes both the DirectMain object and attached deferred objects and creates the MASTER subsystem component. Note that the propagation, once started, occured automatically.

6.4.14 Help Command Executor

The Help command displays information about the command language of the driver. The operands include names of commands, names of user visible variables, and an assortment of miscellaneous keywords.

The HELP system demonstrates two features that should be added to the construction notation. First, named and parameterized rules would prevent the copied construction information. Second, there is no way to specify the connection between component objects and the programs that use them (as required by the "NonResident" version in Figure 6-20).

```
subsystem HELP provides HelpCmdEx
    requires Commands, Variables external CL
    realization
        concrete object Keys=Concat("{Acquire(Commands)}", "{Acquire(Variables)}",
                file(<Miscellaneous Help Keywords>))
        version Resident
            concrete object Pgm=file(<Resident Help Source>),
                Edits=file(<Help Keyword Edits>)
            version Direct select CL=Direct
                resources Concat( "begin ",
                        Edit(BH(Keys) with "sort:A", Edits) with "GenHelpTable",
                        ";", Pgm, " end")
                end Direct
            version Spool select CL=Spool
                resources Concat( "begin ",
                        Edit(BH(Keys) with "sort:A", Edits) with "GenHelpTable",
                        ";", Pgm, " end")
                end Spool
            end Resident
        version NonResident
            concrete object Pgm=file(<NonResident Help Source>)
            version Driver select CL=Direct
                resources Pgm
                components Resolve(Keys)
                end Direct
            version Driver select CL=Spool
                resources Pgm
                components Resolve(Keys)
                end Spool
            end NonResident
        end HELP
```

Figure 6-20: Help Command Executor Subsystem

7. Analysis and Evaluation

We developed a system representation scheme in Chapter 4, used it in the design of a software construction database in Chapter 5 and applied it to parts of a real system in Chapter 6. In this chapter, we evaluate the representation techniques in the light of the experience of the last two chapters. This discussion will address many of the details of this particular representation; general conclusions appear in Chapter 8.

In section 7.1, we discuss many details of the notation, defend choices made in Chapter 4 and suggest improvements for future representations. In section 7.2 we summarize the direct costs of organizing software around a database such as that discussed in Chapter 5. In section 7.3, we discuss indirect effects of organizing software systems with a uniform system description language and central database.

7.1 Basic concepts

In Chapter 4, several decisions were presented without justification. In this section, we discuss many of those points in the representation of basic concepts such as resources, versions and construction rules. In some cases we will explore possible alternatives and either defend the choices we made or indicate a need for a better solution.

7.1.1 Resources

7.1.1.1 Content of Resources and Source. Other researchers have limited the "content" of a resource to programming language constructs such as types, procedures, clusters or variables. We argued in section 3.1.1.1 that extra-linguistic resources were necessary for non-algorithmic information and for abstracting from linguistic features. We exploited this generality to represent program skeletons in section 6.4.10, data collections in section 6.4.6 and construction information in section 6.4.3.

Although no attempt was made to classify source and resource objects in Chapter 4, it is likely that a collection of source object classes will develop. This is because of the advantages of begin able to apply tools uniformly to a class of objects. The objects in Chapter 6 could be grouped into the classes "macros", "editing commands", "source with macro calls", "tables", "Sail programs", "Bliss programs", "Macro-11 programs", "document definitions", "text-oriented documents", and "graphics-oriented documents".

Extending the notion of concrete object to include a "class" would allow the construction system to exploit the class of an object in automatically generating construction sequences. Conventional software construction systems have a fixed set of classes (e.g. a source class for each language, a common relocatable class and a linkage-resolved class). The list above demonstrates that conventional classes are inadequate and we assume that the collections of classes must be extendable by the user. It may also be useful to formulate class as a vector of attributes such that "Fortran macros" and "Sail macros" share the attribute "macros" while "Sail macros" and "Sail source" share the attribute "Sail".

7.1.1.2 Resource Representation. In section 4.2 we restricted the representation of resources to character strings (of arbitrary length). In most system construction processes, character strings are in fact the common currency. Information, however, often exists in structured objects, such as directories, parsed programs, or data bases. In section 6.4.6, a character set list was derived in part from the directory that contains the character set definition files; th

table in Figure 6-11 is created by merging the directory information with auxiliary descriptive text.

Two approaches to this problem are a) to allow non-string resources and b) to provide object-to-string coercion programs (e.g. a directory listing program produces a character string representation of a directory). In the former case, resource manipulation during system construction is more complex because the system must use a different transport operation for each class of object that can be used as a resource. The latter alternative forces the conversion of structured objects to strings before they become resources, and provides them to the requesting site in that canonical form. If the original structure is useful to the requestor, it must be regenerated by parsing the string representation.

Because it is simpler to process canonical string representations, we chose that alternative in Chapter 4. Forcing programs to parse a linear representation to exploit structure that was present before the linear encoding introduces a built-in inefficiency that is probably unacceptable. To our knowledge, only ad hoc solutions to such problems have been developed.

7.1.1.3 Explicit Naming of Resources. The subsystem interconnection notation forces the user to explicitly name all resources. Our experience indicates that this approach is appropriate. For methodological reasons, it should be possible to tell unambiguously what resources are provided and required by each subsystem.

In conventional construction processes there are often associations between the names in one object and those in another, or between the name of an object and names that occur within it. For example, the external symbol names in a compiled code object correspond to identifiers in the program text (more or less), or a macro might be stored in a file with the same name. For another example, resource names in MILs are tied to cluster or function names. Because our resources are character strings, not programming constructs, no implicit association is made between the names of resources and the programming constructs that might be named in the contents of those resources. We believe that the appropriate location for this mapping is in specifications attached to each version of a subsystem. In such specifications, for example, would appear the statement that the resource TrigFcns from a Fortran version of the PLOT subsystem must contain definitions for the Fortran subroutines and functions SetAxis, SetTitle, PlotCircle, PlotFcn and Draw.

7.1.1.4 Structured Resources. We examine in this section whether resources should be hierarchical objects, possibly themselves composed of resources, and whether there should be a capability within the interconnection notation to restrict the use of subresources of a given resource.

Hierarchical organization is a tempting way to organize resources into groups that are required or provided together. This could be used, for example, to define "environments" in which the definition of the environment is composed from the "provides" list of the subsystems that establish the environment. Consider a collection of resources that constitute a mathematical subroutine library. These resources might be provided in an environment definition in a high level subsystem for use throughout the system. If a resource were added to the library, it would be convenient for the environments to be extended automatically to include that new function. A hierarchical resource structure would solve this problem but would be an unnecessarily general solution. The same flexibility can be achieved by the simple textual trick of copying one resource list from another.

Another motivation for hierarchical resource organization is the unification of cases in which one program requires resources such as "Sine", "Cosine" and "ArcTangent", while another program requires "TrigFcns". In Chapter 4, we would provide four separate resources from the same subsystem rather than a tree of resources with "TrigFcns" as the root and the other three as descendants. The relation among them is apparent only from examining the construction of the objects that represent the resources and seeing, for instance, that the object that represents "TrigFcns" was created by concatenating the objects that represented the other three resources.* Again, this benefit does not alone justify the hierarchical mechanism.

If resources are hierarchically structured, either the entry in the provides list must specify the entire tree of resources or the requiring user must specify a path to the resource (we reject searching the entire interconnection structure for the match as inordinately expensive). Suppose we have a resource "MathLib" consisting of "TrigFcns", "SplineFcns", "BesselFcns" and others. If the user can write "Acquire(Sine)" in a program, the library subsystem must specify that "Sine" is available from it directly. As shown by comparison to the current notation, the only advantage in doing so is a suggestive textual format.

subsystem *ML* provides *Sine, Cosine, ArcTangent* . . . . . end *ML*

subsystem *ML* provides *MathLib{TrigFcns{Sine,Cosine,ArcTangent} . . .} . . .* end *ML*

If, on the other hand, we have ML specify only that it provides "MathLib", the user of Sine would have to specify "Acquire(MathLib{TrigFcns{Sine}})". Forcing the user to know the decomposition of resources is unfortunate and in fact negates the value of the scheme in the second example above.

An additional difficulty with hierarchical resources results if the user may write "Acquire(TrigFcns)". In this case *acquire* must be able to construct the higher level resource from the descendant resources. Because composition of resources is dependent on language and program context, some construction rule must be available for *acquire* to use. In Figure 7-1 we show that considering TrigFcns to be a composed resource would alter only the syntax of the first line (as shown) from that imposed by the constraints of Chapter 4.

A related issue is raised by the authors of those interconnection notations that are based on the abstract data type model of a resource. They use a special notation to indicate that only a subset of the operations on the data type are provided or required at some point in a system. Is there a general "subresource" concept that is worthy of a special notation? And if not, what is to be done with the case considered above?

Imposing restrictions on the use of operations from a resource is a means of implementing capability-style protection on objects of the type represented by the resource. Protection is an aspect of the use of resources rather than their definition. We believe that the interconnection notation should describe only definition relationships and that all information about the use of resources, including protection, should be given in the specifications for a resource and the specifications for the use of the resource. To implement these restrictions, additional communication between the specifications and the compiler that enforces them is necessary (see [Tich80]). We recommend that the SCF support this interaction but not be party to it.

---

*The concatenation operator may, of course, be complex. For Algol-like languages, it involves inserting a separator such as a semi-colon, between the definitions.

```
subsystem TRIG provides TrigFcns{Sine*, Cosine*, ArcTangent*}
    subsystem TRIGX provides Sine, Cosine, ArcTangent
        realization
            version Std
                resources Sine: file(<Sine Function Definition>)
                    Cosine: file(<Cosine Function Definition>)
                    ArcTangent: file(<ArcTangent Function Definition>)
                end Std
        end TRIGX
    realization
        version Std select TRIGX=Std
            resources Concat(acquire(Sine), ";",acquire(Cosine), ";",
                    acquire(ArcTangent))
            end Std
    end TRIG
```

Figure 7-1: TRIG--Hierarchical Resource Subsystem

7.1.2 Subsystems

7.1.2.1 Interconnection Mechanisms. The three interconnection mechanisms described in section 4.2, nesting, explicit reference, and environment definition, serve adequately for a large number of system structures. Implicit matching prevents redundancy and keeps the descriptions small.

One result of implicit matching is the potential ambiguity that can arise if resource names are duplicated. While the set of potential providers of resources is explicit, the actual provider is determined by matching against the resource lists of each of the potential providers. Clearly, two subsystems could provide resources with the same name and therefore the choice of providing subsystems is not unique. Because it is likely that resources will be named mnemonically, the duplication of resource names should not be a serious problem. Should it arise, the notation can be trivially modified to resolve the ambiguities: first, qualify the external entries by the resources that they are expected to provide (perhaps using the copying trick proposed in section 7.1.1.3), second, specify the use of resources from nested subsystems in the external clause, and third, specify the use of environmental resources in a separate requires list.

7.1.2.2 Nesting of Subsystems. The distinction between internal subsystems and external subsystems allows some restriction on the scope of names. The need to group objects and encapsulate relationships among them occurs in many contexts and it is no surprise that the same need occurs here. The scoping mechanisms here are similar to those provided by abstract data type languages.

7.1.2.3 Circularity. A programmer can construct circular graphs of subsystems with this notation. Circularity is useful in the case of a subsystem that appears at two levels in a hierarchical system. Suppose, for example, that a high level process resource (e.g. the procedure for creating a process) requires a storage management resource that requires a

low level process resource (e.g. a procedure to halt a process). The apparent circularity in the subsystem interconnection would be necessary to prevent an artificial division in the process subsystem.

Real circularity is unlikely to arise, because system designers do not assume that they solved yesterday the problems they have postponed to today. The ability to represent the kinds of structures that implementors do in fact use is more important than preventing the dangerous structures they are unlikely to exploit. In any case, circularity is easy to detect [Thom76].

7.1.2.4 Scope of Names. The scope of subsystem names was not restricted in the description of the notation (see section 4.2.2.3). In most cases, Algol-like block structured scopes are appropriate, but occasionally it is necessary for a subsystem to name a subsystem nested within a third subsystem. This occurs most often when the using subsystem must independently specify versions for different resources.

We could either require explicit exportation of nested subsystem names or deduce those exportations from the structure of the subsystem. We have chosen to do the latter until we understand better the ramifications of having resources that vary independently even though they are provided by a single subsystem. Because this is a result of having multiple versions of subsystems, the problem does not arise in other interconnection schemes.

7.1.3 Realization Section of Subsystems

7.1.3.1 Separation of Types of Information. The interconnection portion of a subsystem contains only names of resources, names of other subsystems, and subsystem definitions (see section 4.2). The realization section of a subsystem, on the other hand, contains all of the references to tangible objects that "realize" the interconnection structure (see section 4.4). However, within a version in the realization section, we can, as in the interconnection portion, define new subsystems that introduce new resources and name additional external subsystems. It might appear that the careful separation of different types of information has been compromised by this latter ability.

The following argument justifies allowing versions to define subsystems. Some versions of a subsystem have unusual resource requirements. If one version of a subsystem uses a resource that is not used by any other version, then in the absence of the ability to define a subsystem within that version, we must either create a separate subsystem for the version (thereby ignoring the similarities among the versions) or propagate the requirement into the interconnection portion of this subsystem (obscuring the fact that only one version uses it). If the concept of version is an appropriate abstraction within subsystems, then new subsystems must be allowed within versions.

In fact, proper separation of information has been maintained. The information directly within the realization section of a given subsystem implements the interconnections of that subsystem. Any subsystem defined within a version will also be realized within that version.

7.1.3.2 Version Hierarchy. In section 4.4, version names were defined to be vectors of names that described a path through a version hierarchy. Considering those names to be sets of attributes rather than vectors would result in an interesting alternative organization for versions. In such a scheme, the version name "Current.Fortran.Debug" would be equivalent to

"Fortran.Debug.Current". In cases where the attributes are independent of each other, this set characterization is preferable. It avoids unnecessary ordering, encourages "complete" collections of versions (i.e. *all* possible combinations of orthogonal options), and lends itself to efficient specification of construction rules if the attributes are achieved by systematic modification of a single construction rule.

The path interpretation is better if the interpretation of one attribute depends on the selection of another; if the debugging facilities of Fortran and Pascal are dissimilar, "Fortran.Debug" and "Pascal.Debug" do not share construction information. Paths are also useful if there is a natural ordering on otherwise independent attributes. For example, because hierarchical levels include concrete object definitions, one can define higher levels to be those that are likely to contain "source" objects and lower levels to be those that contain "generated" objects. In section 6.4.3 the "current" and "development" sets of control codes are defined in files, whereas the Sail, Bliss and Macro11 versions of any set of control codes are generated by programs. Hierarchical organization provides a convenient locale for each source object that makes it available where it is needed and unavailable elsewhere.

As we learn more about the ways in which versions are used, we will be able to make a clearer choice among these mechanisms for organizing versions. The preceding observations were significant in the choice of path over set in the proposal of Chapter 4.

7.1.4 Versions

7.1.4.1 Resource Objects vs. Component Objects. The two primary types of objects in a version are resource objects and component objects, as discussed in section 4.4.1.3. Is it possible to unify the two classes of objects and simplify the definition of versions?

SCF constructs resource objects on demand but constructs component objects any time a version is built (either directly by the user or indirectly as a side-effect of having one of its resources used). We could achieve this construction timing with the policies of section 5.3.4, so construction timing is not a persuasive argument for differentiating resources and components.

Component objects were introduced because a subsystem may exist without being used by *any other* subsystem. For example, the top level subsystem of any system is used only by external agents. In section 6.4.11, 6.4.7 and 6.4.10 we defined components of such subsystems that contained executable programs or printable documents.

In order to eliminate the "component" object category, we would have to provide an alternative means for the file system to exploit the objects built by SCF. SCF accesses the file system via the *file* operator but we have not defined how a user might, for example, run a program we have built. If we can describe those accesses with strings (e.g. command language text), the file system could interface to the subsystems via the resource mechanism. Suppose that *FileName* is a function that returns the name of the file containing the concrete object that is its parameter. The subsystem in Figure 7-2 illustrates how this might work. Some command language table would have to be responsible for interpreting command language actions in terms of subsystems and versions. For example, the "Scribe" command could be interpreted to mean "execute the command language text in the RunScribe resource of the Current version of subsystem SCRIBE".

It is not certain that this formulation for access to objects in the database is sufficient. However, it will be possible to provide some such interface, and the "component" class of

```
subsystem PROG provides RunProgram
    realization
        version Executable
            resources FileName(Link(Yfpl(file(<Program Source>)))
            end Executable
    end PROG
```

Figure 7-2: PROG--Interface to Operating System via Resources

objects will probably become extraneous.

7.1.4.2 Deferred Objects. The processing of resource objects and component objects by SCF is straightforward and can be discussed easily. Deferred objects, on the other hand, require much more complex mechanisms and descriptive prose.

Deferred objects are the result of multistep construction processes, such as the paradigmic compile/link sequence. If we retain the multistep operation but neglect to enforce the correspondence between the two steps, we will repeat the errors in conventional construction mechanisms. Replacing the link operation with a Mesa-style bind burdens the programmer with specifying the deferred object at a later time (knowing, therefore, that it exists). Avoiding deferred objects by forcing inline compilation, for example, prevents implementors from optimizing the sequence of construction steps.

At the present, we have no reasonable alternative to retaining the linkage to deferred objects. The mechanism is sufficiently awkward, however, that there is likely a useful generalization or a replacement for the compile/link paradigm. In particular, we lose information by collecting deferred objects into an undifferentiated group. There could be deferred objects of several types and it may be necessary to retain information concerning the origin of those objects. Clarifying the nature of deferred objects is a topic for further research.

7.1.4.3 Version Selection. The only means we have introduced for selecting versions is the explicit naming of a version for each relevant subsystem, as described in section 4.4.1.2. This mechanism is adequate (except for environments, see section 7.1.4.5) but often unnecessarily precise. There are situations in which implicit version selection is appropriate, either because the selection is obvious or because any one of several selections is acceptable.

As an example of obvious selection, consider version selection for a resource request from a Fortran program. Certainly we can guess that whenever "Fortran" is a valid selection, it is the correct one. With slightly less confidence, we can make similar assumptions about version choices labeled "Debug" or "Current" or "FredsPrivate". Often "Documentation" and "Backup" will be propagated as well. The programmer should be given a means for defaulting selections or providing sequences of legitimate alternatives at the requesting site and perhaps at the providing site. The latter would allow a subsystem implementor to automatically default to the most commonly used version.

If several versions of a resources are equivalent as far as the user is concerned, it would be incorrect to force the user to be overspecific. For example, a program may be able to use each of an upward compatible sequence of versions and selection from that set should be made on other criteria, such as convenience (from SCF's point of view). Selection from a set

of this type implies that there is a way to define the acceptable subset, i.e. that there are specifications associated with the requirer and each of the provided versions (see section 8.3.1). Once those specifications and the matching facility are defined, selection from a set can be straightforwardly implemented.

7.1.4.4 Additional Provided Resources. Versions of subsystems can contain subsystems that require resources not required by the entire subsystem, as described in section 4.4.1.4. It would be possible to also allow a version of a subsystem to provide resources not provided by other versions.

In the current scheme, a version that does not provide one of the specified resources just omits it from the resources list; each version can in this way provide a subset of a common list. However, suppose that a version could provide a resource of some external subsystem as its own; such a formulation might be appropriate for the definition of the symbol table as a deferred object associated with a special resource for each debugging version.

Exclusion of additional provided resources does not result from an inherent property of subsystems; formally, the provide and require features are difficult to distinguish. Rather, in the view of this author, the capability did not seem appropriate, no need for it was firmly established, and the implementation strategy was so arbitrary that we have postponed consideration of the facility until its role becomes better understood or found to be unnecessary.

7.1.4.5 Environment Definitions. The definition of environments of resources (as introduced in section 4.2.1) add to the complexity of subsystem interconnection semantics. This is reflected in the section of program text that implements the search through environments (see section 5.4.1). Does it contribute proportionally to flexibility? Is the concept correctly represented in the notation?

The basic idea of resource environments is sound, being derived from common practice (see section 1.5); almost every project has a common definitions file that is included in every compilation in the system. It is appropriate that both the providing subsystem and version be specified once for an entire version.

A problem arises in positioning the version selector for an environment. Suppose that two versions of a user system are the executable system and the users' manual for that system. The executable system is implemented in Yfpl using YfplExtensions as a pervasive resource.

```
subsystem SYS provides . . .
    requires YfplExtensions . . . environment YfplExtensions
    external YE
    subsystem SYSA . . . realization . . . end SYSA
    subsystem SYSB . . . realization . . . end SYSB
    realization select YE=Only
        version Exec select SYSA=Exec, SYSB=Exec . . . end Exec
        version Manual select SYSA=Manual, SYSB=Manual . . . end Manual
    end SYS
```

In this case, the specification for subsystem YE applies to both the executable systems and the document systems. If there are two sets of versions, using different versions of YE resources, there is no way to specify the selection without abandoning the environment mechanism. The general problem, of which this is a specific example, is the complete independence of versions, discussed in section 7.1.4.6.

7.1.4.6 Complete Version Independence. If subsystem A has internal subsystems B and C, the versions of B and C are completely independent of each other and of the versions of A (except that A will specify version names for B and C). In practice, however, versions of internal subsystems B and C might be directly related to versions of A; in fact there may exist a partition of the versions such that version A1 uses B1a, B1b, C1a, C1b, etc., while version A2 uses B2a, C2a, etc. Versions of A can exploit their knowledge of versions of B and C but the versions of B and C cannot exploit their membership in the class. An example of this problem was discussed in detail in section 4.5.3. It occurs in a different form in specifying versions of environments (section 7.1.4.5).

Permitting internal subsystem versions to identify with enclosing subsystem versions would cause a substantial reconfiguration of the relationships among resource providers and requirers. No similar problem has been addressed in programming language or operating system research, rendering solution by analogy infeasible. However, such relationships do arise and therefore should be representable as such in the notation.

7.1.4.7 Appropriate Use of Versions and Resources. It is often difficult to decide whether two similar resources are alternate versions of the same information or different resources from the same subsystem. For example, in section 6.4.6 we used the version mechanism to distinguish between the complete and standard character set lists; we could have used a different resource for each.

The criteria for making this decision are similar to those for deciding the representation of portion of a program. If it makes sense to consider the two items as versions, and and the construction details work out, then the version mechanism can be used. If not, then they must be isolated as separate resources. The version alternative does not work if a subsystem needs to use both versions at the same time, or if the version selections for different resources from the same subsystem are incompatible.

When it is necessary to divide a resource into two resources for one of the reasons above, it is often the case that the new resources can be built hierarchically on the old resource. For example, a system that uses two versions of a resource may actually use the same information in two forms; the new resources are then "information in one form" and "information in the other form", which can be built on top of the resource that describes the basic information.

7.1.5 Construction

7.1.5.1 The Acquire Mechanism. One difficulty with the acquire mechanism is that it does not permit a construction process defined in a subsystem to access resources provided by that subsystem. If a subsystem has one version of function that defines callable procedures and another version that incorporates those procedures in an executable program (for example, see the Select program in section 6.4.10), it would be convenient to use the former version of the resources by name. It is not obvious what impact such a capability would have, but it is worth exploration. If it is unduly disruptive to the acquire algorithm, the loss is not great; the same effect can be achieved by defining another subsystem that provides the executable program and uses the resource definitions.

7.1.5.2 Non-transparent Resource Transmission. As defined in section 4.4.1.3, a resource that is both provided and required by a subsystem is passed transparently through that subsystem. An alternative would be to permit the subsystem to systematically modify such resources (an application: addition of measurement information to procedure headers). The semantic modification to the representation is reflected in a minor change in the acquire algorithm; it checks the selected version of the present subsystem for resource objects before looking to the providing subsystem.

The same effect can be accomplished by renaming the resources. This facility trades the flexibility of being able to change resource definitions flexibly against the stability of the meaning of a given resource name in the representation.

7.1.5.3 Functional Rules. The construction rules of section 4.3 are functional in form. We could establish an algorithmic construction language that would permit conditional, iterative or sequential rules. At this time, we have found no need for more general rules and have therefore avoided making them any more complex.

7.1.5.4 Shared Rules. All construction processes are specified by rules, as defined in section 4.3. Rules specify processors and a collection of concrete object and string parameters; each of the concrete object parameters may be another rule. As a result of this definition, it is difficult to express the relationship between two objects which share, among other things, a construction rule.

In a previous version of the structuring notation, there was a mechanism for naming rules for use by several concrete objects. Such a feature is the natural extension of common operating system facilities such as command procedures or job control language files. It was eliminated because it became difficult to define the mechanism with the right degree of generality. The "too little" form allowed only one parameter, the first concrete object slot in the rule. However, shared rules could reasonably be parameterized by each of the parameter slots (both concrete object and string) and even by an intermediate processor name. The "too much" form allowed arbitrary parameterization of rules; the resulting implementation was far too complex for the benefits which accrued.

The conclusion is that construction rules should be given names and permitted parameters. Finding the appropriate formulation of parameterized construction rules is a topic for further research.

7.1.5.5 Side-effect Files. As presented in section 4.3, a processor has only one output object. Yet familiar compilers, for example, generate not only relocatable code but also listings, cross references, statistics, and symbol tables. Other processors produce other side-effect files and some permit the user to produce others. These side-effect files are used primarily as documentation of the primary output but can sometimes be used by other processors; for example, a documentation generator could extract from the listings of programs the storage requirements.

A previous version of this notation allowed for the specification of these auxiliary or side-effect files. However, the complexity of the mechanism exceeded the complexity of the task so the topic has been designated an area for future research. See section 8.3.4 for the basis for this research.

7.1.5.6 Processor Versions. In some environments, the processors themselves exist in several versions, most often a sequence of releases. Dependencies can exist in both directions; a program may be written using compiler features not available until a recent release, and a program may work correctly only on an older compiler version.

We can take a range of approaches to this problem. First, we can consider the processors to be external to the notation and merely provide a way to denote which processor version we desire as a suffix to the processor name. At the other extreme, we could embed the processors in the notation and consider "FortranCompilation" to be a resource provided by a FORTRAN subsystem of which there are several versions. The "outermost" subsystem (analogous to the "outermost" block in an Algol program) would contain subsystem definitions for the various processing capabilities and a resource environment containing the processor resources. A processor name in a construction rule translates to an implicit require of the resource associated with that processor. Any non-standard version selector specified would then override the environmental selector.

7.1.5.7 System Output Objects. Information used to construct a system is also transmitted to the objects created by that system. For example, the Scribe system (see section 6.4.7) requires text-oriented file format information and transmits it into the text-oriented files that it creates. If the TOFF subsystem is changed (see section 6.4.2), the database manager can reconstruct Scribe programs but does not have control over the products of Scribe. In order to keep control of that proliferation of information, every system built within the structuring database must also be a processor in the database. If that is possible (it is not if the system runs on a different computing system), it can be done on an individual case basis.

7.1.5.8 Construction "Uses" vs. Algorithmic "Uses". In section 4.5.3, we found a case in which the notion of "uses" had a different interpretation in an algorithmic context than in a construction context. It is always unfortunate when a concept must be bifurcated, because additional discriminations must be made whenever the concept is employed. Is this problem a result of our formulation or a general property of software systems? Does it result from the generalization of the content of resources?

The particular example of section 4.5.3 is resolved if the version independence problem (see section 7.1.4.6) can be solved adequately. The problem remains, however, as illustrated by an example in which two internal subsystems both use a resource which, for linguistic reasons such as block structure, must be defined only once. In algorithmic terms, each subsystem "uses" the resource independently; in construction terms, the enclosing subsystem must arrange for the definition of the resource and is therefore the constructive "user".

It may be the case that all such examples are the result of archaic language design or poor target system design. However, problems solved linguistically for one domain often recur in another; modern languages carefully provide for abstraction from data representation but fail generally to provide for abstraction from control representation.* Therefore, the nature of this distinction is a topic for further exploration.

---

*Alphard generators [Shaw77] are an exception and help highlight the problem by solving a limited control abstraction with a nontrivial mechanism.

7.1.5.9 Accessing Deferred Objects. The representation currently allows only automatic propagation and explicit processing of deferred objects. This results often in the phrase "Link(x,deferred(x))" where "x" may have been named merely to make the deferred objects accessible. Adding a *deferred* operator to the *acquire* operator on the database would permit processors to access the deferred objects the current object directly. The natural phrase "Link(Yfpl(source))" then works correctly.

There is at present no mechanism for preventing the propagation of deferred objects beyond the point that they are used. If the result of a Link operation is included in some other construction, the now redundant deferred objects continue to be available.

If there are two separate integration processors that use different types of deferred objects, there must be a mechanism for selecting those deferred objects of the relevant type. Because the notion of general deferred objects is not well understood, this mechanism has been postponed for a later version.

7.1.5.10 Access to Component Objects. The association between component objects and the other operating system facilities has not been specified. For example, some component objects are executable programs and must be nameable from the operating system command language. This problem arises in an internal context if a component object is to be used by the resource objects (as required by the "NonResident" version of section 6.4.14).

7.2 Costs Associated with System Structuring

The costs of developing and using a software database of the type discussed in Chapter 5 can be estimated from our exploration of the database implementation.

7.2.1 Development and Implementation Cost

Portions of the system have been implemented and are operational on a PDP-10 computing system. The design and implementation of those sections required approximately six person-months of effort. A database system that included half a dozen processors, policy enforcing manager, and a flexible database entry editor would require two to three person-years of sophisticated programmer time over a period of one to two calendar years. (This estimate assumes that terminal communications, file system, and interprocess communication are available in the computing milieu.)

7.2.2 Storage Costs

7.2.2.1 Database Entry Storage. The space occupied by database entries is proportional to the number of subsystems, versions and concrete objects defined. None of the constituents of these objects become large; an generous average of eight to ten entries in each list is sufficient. If a "unit" in the database is capable of representing a string or a reference to another entry, then the description of a given target system will occupy between 50 and 100 units for each subsystem, version and concrete object defined.

The database manager has some hidden storage in the database containing histories, mailboxes and lists of deferred objects. As mentioned in section 5.2.3.1, histories can become

quite large and they must be migrated periodically. Mailboxes should remain small (zero to 20 units) and can be kept minimal by project policy. Lists of deferred objects should be small and associated only with a small number of other objects.

7.2.2.2 File Storage. The source files for a system are, of course, no larger than in a conventional construction environment. The database organization is designed with the assumption that all source files are immediately accessible, so direct access storage not previously assigned to software source must be allocated.

Because we encourage construction information to be encoded and stored in the file system, use of the database may increase the number of files of "source". This is justified only by the additional reliability that results from mechanically reproducing a set of actions rather than relying on human beings to faithfully duplicate their actions.

An additional load on the file system can result if intermediate construction objects are retained in the file system. Policies can minimize this impact if necessary.

7.2.3 Processing Time

One computing overhead cost results from the invocation of processors by the database manager rather than the human programmer. The extent (and sign) of this cost is dependent on operating system design. There are mechanisms in some operating systems that permit programs to invoke other programs more quickly than a user can. In other systems, there is no easy way to perform the task at all, especially if the calling program must retain its state across the call.

Most of the additional computing cost results from the increased ability to decompose problems more thoroughly. For example, it is now feasible to maintain a program which is constructed in five steps; the manager can guarantee that the necessary constructions have been performed whenever that program is used. Without such a facility, it is likely that the five step construction would have been abandoned because it was too risky.

The greatest potential for increased computation is in the profligate use of automatic construction actions in policies. Some overhead is necessary, namely the propagation of messages to objects. This overhead is, however, much lower than the alternative in which each construction must examine each low level component to determine if a change has occured. The project management can regulate the amount of automatic computation performed by careful organization of policies.

7.2.4 Cost Reductions

Some of the database manager facilities reduce costs. Storage space is gained by encouraging users to share rather than duplicate information ,and to highly encode information. Explicit usage histories provide the information needed to purge obsolete objects immediately. Policies associated with intermediate objects can cause automatic deletion of objects not likely to be used again soon.

Computation costs are reduced because processing steps are never duplicated (unless the result has been discarded). The subdivision of computation into separate steps permits some computation to be saved even if other components are being modified.

7.3 The Effects of Structuring

7.3.1 Centralization vs. Control

The organization of software development around a central database results in a centralization of the project and increased control by project management over the process of construction. How do the proposals we have made minimize the conflict over centralized resources, permit flexibility on the part of individuals, and maintain management control without sacrificing creativity by project members?

The problems caused by centralization are not alleviated in any way by the proposals put forth here. Dependence on a central facility that probably exists on a central computing system is an unfortunate result. While we can hypothesize either that the database can be partitioned with only occasional boundary crossings, or that research in distributed computing will result in transparently distributed databases, we must realistically admit that there will be restrictions. Software construction will depend on the availability of the database. In installations with single processors and uniformly addressable storage, of course, the database is available whenever the computing system is available.

The problem of availability is particularly difficult in the case of systems that are transferred from one computing system to another. Although we have only improved distribution techniques by automating the bookkeeping on system components, the distributed system cannot be maintained in the field with the same flexibility unless the field installation has a similar database into which the system descriptions can be entered.

The flexibility of programmers is not impeded if the programmer for a given subsystem has complete control over the subsystem database entry. Only rigidity in the construction notation will limit the programmers' ability to use the installation facilities.

Management control increases because the database manager can be used to control access to both files and subsystem descriptions, as well as to enforce policies about system consistency. Although management control conflicts with programmer flexibility, the database manager can be used to enforce whatever level of control management deems appropriate.

7.3.2 Separation of Function

In conventional construction environments, language compilers are used to perform construction and editing tasks in addition to language translation *per se*. The source text is processed by only one program (although it may make several passes in order to accomplish all of the tasks). In an environment that supports highly decomposed and automated software construction, some source information is processed by several different processors. Those processors that operate sequentially on information must either communicate with each other or partially duplicate computation (e.g. lexical analysis). Channeling information from one processor to another becomes a problem in itself.

The transmission problem exists in traditional environments between compilers and linkers. Generally the communication is accomplished though the file system with a special compiler-to-linker format. If there are several processors, having a separate protocol for communication between each pair is an unreasonable solution. Therefore, we suggest a common processor-to-processor channel language in which one can encode a variety of types of information. This language is analogous to the communication protocol on a hardware link.

The communication protocol provides the ability for processes to send messages across the link without being concerned about how messages are terminated and what other processes are using the link. By standardizing the representation of parsed text, the channel language would facilitate the integration of such programs as cross-reference generators, indenting formatters, macro expanders and compilers.

Some of the excess text processing can be avoided in this way, but in general, increasing the variety of processors which manipulate information will result in redundant processing of source. In particular, if processors operate in sequence, each will read the results of the previous one. Concurrent organization of the processors is a possible research topic [Krut75].

7.3.3 Installation Requirements

The following sections discuss the requirements placed on the operating system environment by a software database system.

7.3.3.1 Interactive Terminal Communications. SCF must be able to control modification and access to the database. It must therefore be able to determine the identities of users. Also, because users access the database simultaneously, SCF must have synchronization mechanisms available that can impede the progress of one user due to the actions of another. Other than these, no special communications techniques are necessary.

7.3.3.2 File System Facilities. The file system for software development need not be as complex as that for a general utility system [Dolo76a]. Most construction processes are sequential so sequential files are generally adequate. However, the contents of these files should be structured so that several types of information can exist in the same file. An architecture that precedes each variable length block of data with a count and a type field permits easy implementation of such techniques as delta-style maintenance of sequential changes to files, extraction of portions of files, and protection of private portions of file contents [Reid77, Roch74].

Some of the concepts developed for concrete objects are more correctly associated with files. For example, a compiler creates both a relocatable program and listing from a source object. This pair of objects are actually two aspects of a single "result" object. No listing would ever be associated correctly with any other relocatable program. In general, the concept of file must be broadened to include several types of information [Habe77].

7.3.3.3 Processor Design. An integrated software database system requires that processors such as compilers and editors not attempt to preempt the construction processes. All use of file and directory system information should be deleted and replaced by uniform invocations of SCF, such as with the *acquire* function of section 4.4.4. Removal of macro and conditional compilation features from processors can significantly improve their performance while properly placing such facilities in a text processing environment.

Use of externally provided resources is facilitated if processor design does not place unnecessary restrictions on resource format or placement. Order of definition should be irrelevant, as should line length, spacing, indentation, and depth of nesting. Flexibility of usage (expression for statement and vice-versa) and bizarre syntax (null expressions and statements, unnecessary block brackets) need to be tolerated. Many issues that are important for analyzing programs written by people (e.g. indentation) are trivial for

analyzing programs written by other programs.

7.3.3.4 Processor Interface. The ability to use processors as subroutines considerably eases the implementation of SCF. This feature, combined with the ability to have several communicating parallel processes accessing the same data base, allows SCF to operate in an environment in which it is sovereign. This results in significant advantages in efficiency, consistency, failure recovery, and flaw-repair.

Not all operating systems permit easy use of processors as subroutines. In OS/360 systems, for example, one mechanism works only if both caller and called program share a memory region and do not interfere with each other. TOPS-10 monitors require that the called process be established as a pseudo-terminal and controlled by a program simulating a human user. On the other hand, UNIX systems provide a flexible mechanism for interconnecting programs into pipelines; this mechanism is well suited to our needs.

8. Conclusions and Directions

8.1 Results

The primary goal of this work was to develop a representation for software systems that integrated design and construction information and incorporated the concept of multiple-version systems in a reasonable manner. The representation we have developed improves on the state of the art in several ways.

First, the generalization of construction processes and the content of resources permits our representation to apply to systems with a wide variety of automated construction techniques. The single step construction process (e.g. compilation) can be replaced by the more complex construction processes currently typified by the construction of compilers using parser generators. The semantic rigor associated with programming languages will soon extend to other construction tools and can be easily exploited within the same framework.

Second, we succeeded in coordinating the use of information among the versions of systems. The degree of shared information and the points of deviation are directly associated with structural features of the system description. With a small number of mechanisms, we were able to describe the relationships between versions that differ along several dimensions.

The feasibility of integrating design and construction information was demonstrated by the examples that showed the actual construction of system components. The interaction between the processors and the software construction database actually results in control of the construction process by the design description.

Finally, we have contributed to the arguments for organizing software construction around a unified database. The contributions by others have emphasized the managerial and bookkeeping advantages derived from such a database; we have added tools that improve the flexibility, reliability and consistency of the software itself.

8.2 Conclusions

The design/implementation process can be significantly shortened by encoding construction processes rather than performing them directly. Therefore, the iteration of design can be emphasized because the implementation techniques have been recorded and can be reused and/or modified. The usefulness of this technique can exceed its cost only if correct construction order and bookkeeping are automatically controlled.

Rigidity of system design can likewise be reduced. First, the design information is recorded in a manner that permits programmers who did not design the system to learn its structure and to manipulate local sections of the design description. Second, the automation of construction processes permits a reconfiguration of the system with less effort on the part of the programmer.

Techniques for constructing families of systems have not developed in the past due to the difficulty of maintaining parallel versions of systems. Although many low level tools must also be engineered to make family construction practical, the framework we have provided can make those tools useful and therefore make family construction practical.

The integration of more and more information about a system into a central, unified database permits the development of more powerful management tools. Control of resources,

such as storage space and computing time, can be extended and in many cases automatically performed. For example, tradeoffs between storage and recomputation of intermediate results can be adjusted and enforced automatically without requiring attention by individual staff members.

It is not known what level of sophistication the technical management and staff of a project must attain to make use of the techniques described here. It is likely that at least the technical management must be highly sophisticated in the use of software tools; it is hoped that corresponding levels of sophistication are necessary for staff usage of the mechanisms.

8.3 Topics for Further Research

8.3.1 Specifications

Individual subsystems and versions of subsystems serve purposes that may be encoded in specifications. Some of these are input/output specifications that describe the functional behavior of programs; others dictate form, format, space requirements, side-effects, resource usage, and other restrictions or assumptions. To integrate specifications into the representation, we must develop a method for encoding general specifications and matching requirement specifications with provision specifications. Among the issues to be addressed in this matching scheme are compatibility, upward compatibility, functional equivalence, minimal satisfaction, uniformity, and type.

In section 4.5.1, we created a set of objects from the terminal type table of Table 4-29. Some of the specifications associated with objects in that subsystem are listed here:

- TerminalTypes: an encoding of the set of terminal types from one of two tables such as that shown in Table 4-29.

- Identifiers1: provides the resource TerminalTypes as a Yfpl declaration using the first of the two sets of terminal types. For table entry with three character code "xxx" there will be an identifier "Ttxxx" that evaluates to an integer with the corresponding binary value.

- Indentifiers2: same as Identifiers1 using the second set of terminal types.

- Vectors1: provides the resource TerminalTypes as a Yfpl declaration. "TtBitpattern" and "TtNames" will be arrays with index range [1..TtNum] (where TtNum is a Yfpl identifier that evaluates to an integer). The i-th entry of TtBitpattern will contain an integer corresponding to the string description in the i-th entry of TtNames. All table entries will represented in the vectors, but order is irrelevant.

- Vectors2: Same as Vectors1 using the second set of terminal types.

- TrmSets: each page of this file contains data such as that in Figure 4-29 as a series of lines with columns separated by tab characters. The first column contains exactly three characters, and each entry in the column is unique. The second column contains a binary constant of at most (wordsize) bits; these must also be unique in the column. The third column contains a string unique within the column.

- Set1: Contains the first set of terminal types in the format described for TrmSets.

- Set2: Same as Set1 using the second set of terminal types.

- TrmEdits: Contains Snobol4 functions called TabToId and TabToVec that will convert a table of the form described in the previous paragraph to the forms described above if executed in the context of the Edit program (specifications by *reference*).

Specifications are usually in part implicit. Because human beings make assumptions about software objects, many requirements need not be stated. For purposes of this work, specifications must be adequate only to discriminate among versions of systems and to group "compatible" versions. Part of this research should exploit the recent progress in requirements analysis to determine whether those methods are sufficiently general for describing the requirements of non-procedural objects.

8.3.2 Program Generation Techniques

One approach to higher level programming is the development of higher level programming languages that construct algorithms from input/output specifications or optimize programs written in a notation without data structures by selecting appropriate representations. These projects have often gone under the heading "automatic programming"; they derive their results primarily by exploiting the semantic structure of programming concepts such as iteration, sequencing, and assignment. An alternative is to write programs that translate other notations into programs by using information about the domain of the problem. Examples include parser generators, code generator generators[Catt78], and decision table translators.

With the current state of the art, both of these techniques require high initial investment of effort. The automatic programming approach is being explored profitably by Balzer [Balz76] and others. A single facility for performing this task is all that is needed because programming concepts are independent of domain.

The translation approach to program generation needs further analysis. In order to make translation of notations into programs a commonplace technique for software engineers, we must develop tools for building such translators. Techniques for parsing, table building, interpreting, high level string handling (e.g. build set of parallel vectors in programming language X, where X is a parameter containing the relevant sections of the grammar), and syntax-directed editing should be encoded in a tool building library. The next order of magnitude of programmer productivity will come not by providing implementations of common low level programs (e.g. list packages, stacks, symbol tables), but by encoding the methods by which information from a problem is encoded in programs.

8.3.3 Representation Details

The following aspects of the representation and the corresponding processing by SCF need additional attention.

- The specification, naming and parameterization of construction rules. Construction rules should be sharable, parameterized in both the concrete object

and string parameter portions, and possibly available as resources. If possible, integrating construction rules into the concrete object realm would unify the representation of construction information and permit the use of system building tools on the construction rule representations. In particular, processors can be defined as subsystems in the database that provide resources that contain (parameterized) construction rules. Such a formulation allows SCF to have a minimum of built-in information about individual processors and permits the user to extend the set of processors easily.

- Alternate methods of determining version, such as default selection, automatic selection, and specification matching. The arguments and suggested directions were discussed in section 8.3.1.

- Handling of construction side effect files, such as listings, maps, and statistics. The output of a processor can be considered an object with several fields; in the case of compilation the fields are "compiled code", "printable listing", "cross-reference information", "compiler statistics", "program statistics", "debugger symbol table", "user generated files" and so forth. Rather than make a special case of processor output, however, we recommend developing a generalization of the concept of file, as discussed in section 8.3.4 and using that for representing processor output.

- Handling of construction failures (e.g. compilation errors). One of the strongest arguments for a more general construction language comes from the need to test the results of processors. If processors had only success or failure results, failure would always propagate failure and construction would terminate. However, systems are often built and tested with unresolved external symbols, compiler warnings and other intermediate error conditions. Therefore the construction language must provide the user the ability to state what conditions will terminate construction and what conditions may be ignored or result only in warnings. If possible, this mechanism should be incorporated in the functional framework as with a "maximum permitted error level" indication in each construction rule.

- Primitives for use by policies and query systems. In Chapter 5 we hypothesized a set of requirements for the primitives of a database command language and query language, and suggested that the policy language would be a composite of the two. These hypotheses need validation by the implementation of a full, production quality command and query language.

8.3.4 File Structures for Software Construction

The notion of concrete object, as developed here, may serve as the basis for a file structure that is suitable for software construction. It is especially promising to consider a file to be a typed object with potentially several fields containing portions of the object. For example, the output object of a compiler contains separate fields for the listing, relocatable machine code, cross reference, debugger symbol table and accounting information.

The simple file is analogous to a "scalar" in a programming language, and it makes sense to have several scalar file types (e.g. ASCII text, machine code). In addition to the record-style structure of the above example, an array structure is useful for sequential version files such as those maintained by the Source Code Control System [Roch74].

The questions to explore in this analogy to data structuring include the following:

1. What structuring methods, in addition to records and arrays, are appropriate for use with files?

2. Does it make sense to have components of structured files be themselves structured files? If so, what are the semantics of such constructs? What is the expense of implementing such a general structuring capability?

3. Does the notion of pointer play a role in file structuring (outside of database applications)?

4. The field names of records could be made static, forcing a type definition construct like that of PASCAL, or dynamic with the resultant "run-time" interpretation. Since we are dealing with files, not simple variables, even a simple dynamic scheme would be relatively inexpensive.

8.3.5 Database Issues

The monolithic nature of the software database in Chapter 5 is a disadvantage for environments in which there are natural divisions between systems, either technical (distributed computing systems) or managerial (security, accounting, or arbitrary). We should explore the feasibility of dividing the database and establishing mechanisms for communicating between the SCF processes of the resulting sections.

Most of the problems associated with distributing the software construction database are general database problems and are not made more or less difficult by the application. There are some comments that we can make from the special features of this application.

1. The hierarchical structure of objects in the database makes subdivision convenient along one dimension and difficult along all others. Therefore, we expect that a section of the database will include one or more independent subsystems and all of their associated versions and concrete objects.

2. The direct communication between subsystems is already channeled through *acquire*. Therefore, the communication between sections of the database for direct transfer of information will be contained within the implementation of *acquire*.

3. Indirect communication between subsystems includes messages, interrogation of conditions by policies and synchronization of construction processes. The first two can be implemented in the message and policy primitives; the latter will require additional programming in the SCF process scheduler.

4. If all processing programs needed by a given section of the database are available to it, no additional work is necessary to accomplish construction processing. In that case, each subsystem will have its own SCF process which schedules and invokes processor for local operations. However, if some processors are available only under control of a subset of the SCF environments, then additional programming is required to coordinate processing requests.

We believe that managerial constraints such as access and modification control can be enforced with existing protection technology associated with database objects. It is possible, however, that the database objects do not directly correspond to the domains of managerial concern, and that additional mechanism will have to be built into SCF.

8.3.6 Programming Language and Compiler Design

The full value of a software construction facility is attainable only when the various software tools and the construction environment are well integrated. Although we have avoided assuming much about the languages and compilers that are available, we intend that the system can effectively exploit them. There are several lines to explore in this regard.

How can compiler information be expressed for use by general utilities? One aspect, the grammar for the language, is easily encoded for use by syntax-oriented editors, preprocessors, and so forth. Axioms for various language constructs can also be encoded for use by a verifier or test case generator. Some necessary compiler information is more mundane. For example, we may need to know the maximum identifier length, line length, or nesting level permitted by the compiler. We also may wish to know the manner in which one initializes an array, the order in which declarations may appear or the relative efficiency of procedure calls and storage allocation. At the other end of the spectrum are installation programming practices and conventions, strategies for good use of the language, and knowledge of the program library.

In the preceding paragraph, we discussed extracting compiler information for use by other tools. We might also consider how one would design a compiler assuming that it would be used primarily or only as a subprogram of an SCF. Some of these issues were mentioned in section 7.3.3.3.

A compiler that is designed for use in a flexible construction environment would make it easy to construct and process fragments and skeletons of programs. For example, suppose that the Y version of resource X is specified to be a statement in Yfpl. The Yfpl complier should be able to parse X (even though it is not a complete program) and assert that it is or is not a statement. It should also be able to parse a user of X by assuming that X is a statement without seeing it.

In order that programs that generate programs be as simple as possible (there may be many of them) the language should have as few restrictions as possible on order, length, redundancy, format, depth, embedding or binding time. An restrictions motivated by the inappropriateness or unlikelihood of people writing such programs must be abandoned or made overridable.

To the degree that implementation flexibility can be handled directly within the language, it is fine to do so. But at the point that the language begins to address extraneous issues such as text or file manipulation, is should desist and leave it to the SCF. Most importantly it should not get in the way of other tools that attempt to provide extra-linguistic flexibility. Bad precedents include the internal editors of APL or LISP, the simplistic syntax of PASCAL, the complex internal states of some interactive programming tools, and non-text representation of data that could be represented by text.

8.3.7 The Difficulty of Multiple Abstraction

Why do programmers have such difficulty abstracting from programs? The following

hypotheses deserve exploration. With a solid understanding of the actual problems encountered by programmers in multiple level abstraction, we can develop methods that expand their abilities to do it.

- Since abstraction is a difficult task, there is resistance to additional abstraction; this manifests itself in the reluctance of programmers to formulate mathematical models of their programs or to specify verification conditions.

- The abstraction from programs is on a higher level than abstraction from the domain, so there is the additional problem of maintaining the boundary between the two abstractions.

- The tools available for performing abstraction from program objects are inadequate and often very poorly engineered. Even well motivated programmers become discouraged from doing this second level of abstraction when they find that they will be attempting to sculpt in granite with a pocket knife in one hand and a jackhammer in the other.

8.3.8 Interactions with Other Technologies

In other approaches to program construction, the construction processes are partially embedded in the programming language system[Gesc77, Wulf76, Lisk76]. We have recommended that programming languages stay within the algorithmic domain. Resolution of this conflict depends on better understanding of the problems being solved by the other systems, determining whether those same problems can be solved elegantly within our framework, and in finding the real limitations to embedded schemes (as opposed to the limitations upon the current implementations of such schemes).

IV Appendices and References

I. Edit Global Facilities

I.1 Text Utilities

```
*****************************************************************
*                                                               *
*               Useful identifier defs                          *
*                                                               *
*****************************************************************

                uc = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                lc = "abcdefghijklmnopqrstuvwxyz"
                letters = uc lc
                numerals = digits = "0123456789"
                alphanums = letters  numerals
                cr = ASCII(15)
                lf = ASCII(12)
                crlf = cr lf
                tab = ht = ASCII(11)
                ff = ASCII(14)
                vt = ASCII(13)
                bs = ctlh = ASCII(10)
                ctlu = ASCII(25)
                bell = ctlg = ASCII(7)
                esc = ASCII(33)
                rubout = ASCII(177)



*****************************************************************
*                                                               *
*               Case folding routines                           *
*                                                               *
*****************************************************************

                DEFINE("UPPER(string)")                 :(UPPER.end)
UPPER           UPPER = REPLACE(string,lc,uc)   :(RETURN)
UPPER.end

                DEFINE("lower(string)")                 :(lower.end)
lower           lower = REPLACE(string,uc,lc)           :(RETURN)
lower.end


*****************************************************************
*                                                               *
*               Simple Text Operations                          *
*                                                               *
*****************************************************************

                DEFINE("TRIMX(str)")    :(TRIMX.end)
TRIMX           TRIMX = TRIM(str)
                TRIMX FENCE SPAN(" ") =         :(RETURN)
TRIMX.end


*       Replace all occurances of S1 with S2 in STR

                DEFINE("SUBSTITUTE(str,s1,s2)s,t")      :(SUBSTITUTE.end)
SUBSTITUTE      s1 LEN(1) . s                   :F<3>
                str FENCE BREAKX(s) . t s1 =    :F<2>
```

```
                SUBSTITUTE = SUBSTITUTE t s2    :1>
                SUBSTITUTE = SUBSTITUTE str     :(RETURN)
SUBSTITUTE.end
```

I.2 Snobol Code Generation Utilities

```
        DEFINE("SnobolRequire(x)")      :(SnobolRequireEnd)
SnobolRequire
        SnobolRequire = "REQUIRE(" QUOTE(x) ")" :(RETURN)
SnobolRequireEnd

        DEFINE("SnobolFcn(name,parms,body,locals)line1")        :(SnobolFcnEnd)
SnobolFcn
        body FENCE crlf =       :S<>
        body crlf RPOS(0) =     :S<>
        body FENCE tab = name tab       :S<2>
        line1 = name crlf
        SnobolFcn = tab 'DEFINE("' name '(' parms ')' locals '")' tab ':(' name 'End)' crlf
+               line1 body crlf name 'End' crlf crlf    :(RETURN)
SnobolFcnEnd

        DEFINE("SnobolStmt(stmt,label,goto)")   :(SnobolStmtEnd)
SnobolStmt
        SnobolStmt = label tab stmt tab goto crlf       :(RETURN)
SnobolStmtEnd

        DEFINE("SnobolCont(line,goto)") :S(SnobolContEnd)
SnobolCont
        SnobolCont = "+" tab tab line tab goto crlf     :(RETURN)
SnobolContEnd
```

II. Text-Oriented File Format Macros

II.1 File: Sail Toff Gen Macros

```
****************************************************************
*                                                              *
*              Sail Version Of ToffGenerate Resource           *
*                                                              *
****************************************************************

        DEFINE("ToffText(str)")                 :(ToffTextEnd)
ToffText
        ToffText = " Begin String Sdum;Sdum←" str
+               ";While Length(Sdum)>0 Do OutputAppend(Lop(Sdum)) End "         :(RETURN)
ToffTextEnd

        DEFINE("ToffCtls(str)")                 :(ToffCtlsEnd)
ToffCtls
        ToffCtls = " Begin String Sdum;Sdum ← " str ";OutputAppend(0);"
+               "OutputAppend((Length(Sdum) AND '177400) LSH -7);"
+               "OutputAppend(Length(Sdum) AND '177);"
+               "While Length(Sdum)>0 Do OutputAppend(Lop(Sdum)) End "          :(RETURN)
ToffCtlsEnd
```

II.2 File: Sail Toff Dcd Macros

```
****************************************************************
*                                                              *
*              Sail Version of ToffDecode Resource             *
*                                                              *
****************************************************************

        DEFINE("ToffNext(str,success,fail)")    :(ToffNextEnd)
ToffNext
        ToffNext = " Begin Define ToffText=False,ToffCtls=True;"
+               "String ToffUnit;Boolean ToffGotIt,ToffType;"
+               "Integer ToffPtr,ToffCnt;"
+               "External Integer Procedure Index(string x,y);"
+               "ToffGotIt←False;"
+               "If Length(" str ")>0 Then If " str "[1 For 1]=0 Then "
+               "Begin If Length(" str ")>1 Then Begin ToffUnit←Lop(" str ");"
+               "ToffCnt←(Lop(" str ") LSH 7) LOR Lop(" str ");"
+               "If Length(" str ")≥ToffCnt Then "
+               "Begin ToffUnit←" str "[1 To ToffCnt];"
+               str "←" str "[ToffCnt To ∞];"
+               "ToffType←ToffCtls;ToffGotIt←True End End End "
+               "Else If (ToffPtr←Index(" str ",0))>0 Then "
+               "Begin ToffUnit←" str "[1 For ToffPtr-1];"
+               str "←" str "[ToffPtr to ∞];"
+               "ToffType←ToffText;ToffGotIt←True End;"
+               "If ToffGotIt Then " success " Else " fail " End "      :(RETURN)
ToffNextEnd

        DEFINE("ToffPartition(act1,act2)actpat,acttext,actctls")        :(ToffPartitionEnd)
ToffPartition
        actpat = ("text:" REM . acttext) | ("ctls:" REM . actctls)
        act1 actpat     :F(Error)
        act2 actpat     :F(Error)
        ToffPartition = "If ToffType=ToffText Then " acttext " Else " actctls   :(RETURN)
ToffPartitionEnd
```

III. Control Code Subsystem Source Files

III.1 File: Sail CC Edits

```
*****************************************************************
*                                                               *
*               CcDefinition Parsing Pattern                    *
*                                                               *
*****************************************************************

        CcPattern = FENCE BREAK(tab) . CcCode tab
+               BREAK(tab) . CcValue tab
+               ("Integers1" | "Integers2" | "String" | NULL) . CcParm tab
+               BREAK(tab) . CcInit tab
+               REM . CcDescription

*****************************************************************
*                                                               *
*               CcGenerateSail Edit                             *
*                                                               *
*****************************************************************

        CcTab = TABLE(3)
        CcTab<'Integers1'> = '"&" A'
        CcTab<'Integers2'> = '"&((" A " LAND ' "" '177400) LSH -7)&(" A " LAND ' "" '377)"'
        CcTab<'String'> = '"&((Length(" A ") LAND ' ""
+               '177400) LSH -7)&(Length(" A ") LAND ' "" '377)& " A '

        DEFINE("CcGenerateSail()t1,t2")         :(CcGenerateSailEnd)
CcGenerateSail
CcGenSailLoop   EditInput CcPattern     :F(RETURN)
        t1 =
        t1 = DIFFER(CcParm) "A"
        t2 =
        t2 = DIFFER(CcParm) CcTab<CcParm>
        EditOutput = SnobolFcn('Cc' CcCode,t1,
+               SnobolStmt('Cc' CcCode ' = ' QUOTE(CcValue) " " t2)
+               SnobolCont(,':(RETURN)'))
+                                       :(CcGenSailLoop)
CcGenerateSailEnd

*****************************************************************
*                                                               *
*               CcDecodeSail Edit                               *
*                                                               *
*****************************************************************

        DEFINE("CcDecodeSail()")                :(CcDecodeSailEnd)
CcDecodeSail
        CcNextSail()
        REWIND(input)
        CcPartitionSail()       :(RETURN)
CcDecodeSailEnd


        CcNextTab = TABLE(3)
        CcNextTab<'Integers1'> = '" If Length(Sdum)>0 Then CcParmI←Lop(Sdum) Else CcErr←True"'
        CcNextTab<'Integers2'> = '" If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "' crlf
+               '+' tab tab '"LOR Lop(Sdum) Else CcErr←True "'
        CcNextTab<'String'> = '" If Length(Sdum)≤1 Then CcErr←True Else Begin "' crlf
+               '+' tab tab '"CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);' crlf
+               '+' tab tab 'If Length(Sdum)<CcParmI Then CcErr←True Else "' crlf
+               '+' tab tab '"Begin CcParmS←Sdum[1 TO CcParmI];'
```

```
+               'Sdum←Sdum(CcParmI+1 TO ∞) End End "'

        DEFINE("CcNextSail()p,sep,b")                   :(CcNextSailEnd)
CcNextSail
        b = SnobolStmt('CcNext = ' QUOTE(" Begin Boolean CcErr;Integer CcCode, CcParmI;"))
        b = b SnobolCont(QUOTE("String CcParmS, Sdum;CcErr←False; Sdum←") ' ctls ' QUOTE(";"))
        b = b SnobolCont(QUOTE("If Length(Sdum)=0 Then ") ' terminate ')
        b = b SnobolCont(QUOTE(" Else Begin CcCode←Lop(Sdum);"))
        b = b SnobolCont(QUOTE('Case CcCode of Begin '))
        sep = ' '
CcNextSailLoop  EditInput CcPattern             :F(CcNextSailDone)
        p =
        p = DIFFER(CcParm) CcNextTab<CcParm>
        b = b SnobolStmt('CcNext = CcNext ' '"' sep '(' CcValue ') " ' p)
        sep = ';'       :(CcNextSailLoop)
CcNextSailDone  b = b SnobolStmt('CcNext = CcNext " End;"')
        b = b SnobolStmt('CcNext = CcNext "If NOT CcErr Then " succeed '
+               '" Else " fail " End End "',,':(RETURN)')
        EditOutput = SnobolFcn("CcNext","ctls,succeed,terminate,fail",b)        :(RETURN)
CcNextSailEnd

        DEFINE("CcPartitionSail()b,p")                  :(CcPartitionSailEnd)
CcPartitionSail b = SnobolStmt('CcTable = TABLE(38)')
        b = b SnobolStmt('DATA("CcData(value,parm)")')
        b = b SnobolStmt('CcActionPattern = BREAK(":") . CcCode ":"
+               ' BAL . CcAction ("," | RPOS(0))')
CcPartSailLoop1 EditInput CcPattern             :F(CcPartSailAct)
        CcParm "Integer" REM = "CcParmI"
        CcParm "String" = "CcParmS"
        b = b SnobolStmt('CcTable<' QUOTE(CcCode) '> = CcData('
+               CcValue ',' QUOTE(CcParm) ')') :(CcPartSailLoop1)
CcPartSailAct
        b = b SnobolStmt('CcSep =')
        b = b SnobolStmt('CcPartition = " Case CcCode of Begin "')
        b = b SnobolStmt('actions CcActionPattern =',',Loop1',':F(Done)')
        b = b SnobolStmt('CcCode = TRIMX(CcCode)')
        b = b SnobolStmt('CcDefault = IDENT(CcCode,"e") CcAction  :S(Loop1)')
        b = b SnobolStmt('CcParm = parm(CcTable<CcCode>)')
        b = b SnobolStmt('p =')
        b = b SnobolStmt('p = DIFFER(CcParm) ' '"' '(" "' ' CcParm ' '" ")' '"')
        b = b SnobolStmt('CcAction "cg>" = "!Cc" CcCode p "!"')
        b = b SnobolStmt('CcAction "cp>" = CcParm',',Loop2',':S(Loop2)')
        b = b SnobolStmt('CcPartition = CcPartition CcSep " [" '
+               'value(CcTable<CcCode>) "] " CcAction')
        b = b SnobolStmt('CcSep = ";"')
        b = b SnobolStmt('CcTable<CcCode> =',,':(Loop1)')
        b = b SnobolStmt('DIFFER(CcDefault)','Done',':F(Exit)')
        b = b SnobolStmt('CcArray = CONVERT(CcTable,"ARRAY")',,':F(Exit)')
        b = b SnobolStmt('I = 1')
        b = b SnobolStmt('CcCode = CcArray<I,1>','Loop3',':F(Exit)')
        b = b SnobolStmt('CcParm = parm(CcTable<CcCode>)')
        b = b SnobolStmt('CcAction = CcDefault')
        b = b SnobolStmt('p =')
        b = b SnobolStmt('p = DIFFER(CcParm) ' '"' '(" "' ' CcParm ' '" ")' '"')
        b = b SnobolStmt('CcAction "cg>" = "!Cc" CcCode p "!"')
        b = b SnobolStmt('CcAction "cp>" = CcParm','Loop4',':S(Loop4)')
        b = b SnobolStmt('CcPartition = CcPartition CcSep " [" '
+               'value(CcArray<I,2>) "] " CcAction')
        b = b SnobolStmt('CcSep = ";"')
        b = b SnobolStmt('I = I + 1',,':(Loop3)')
        b = b SnobolStmt('CcPartition = CcPartition " End"','Exit',':(RETURN)')
        EditOutput = SnobolFcn('CcPartition','actions',b,'CcSep,p,1')   :(RETURN)
CcPartitionSailEnd

*****************************************************************
```

```
*                                                                     *
*               CcStateSetSail Edit                                   *
*                                                                     *
***********************************************************************

        DEFINE("CcStateSetSail()b,sep")         :S(CcStateSetSailEnd)
CcStateSetSail
CcStateLoop     EditInput CcPattern             :F(CcStateDone)
        b = b DIFFER(CcInit) SnobolStmt('CcStateSet = CcStateSet ' sep
+               'CcStateSub(template,' QUOTE(CcCode) ',' CcValue ','
+               QUOTE(CcParm) ',' QUOTE(CcInit) ')' )
+               ?(sep = "separator ")   :(CcStateLoop)
CcStateDone     b = b SnobolStmt(,,":(RETURN)")
        EditOutput = SnobolFcn("CcStateSet","template,separator",b)
        EditOutput = SnobolFcn("CcStateSub","template,code,value,parm,init",
+               SnobolStmt('parm "*" REM =')
+               SnobolStmt('template "cc>" = code','CcStateSub1',':S(CcStateSub1)' )
+               SnobolStmt('template "cv>" = value','CcStateSub2',':S(CcStateSub2)' )
+               SnobolStmt('template "ct>" = parm','CcStateSub3',':S(CcStateSub3)' )
+               SnobolStmt('template "ci>" = init','CcStateSub4',':S(CcStateSub4)' )
+               SnobolStmt('template "cg<>" = "!Cc" code ' "'" ')(" "',
+                       'CcStateSub5',':S(CcStateSub5)' )
+               SnobolStmt('template "cg>>" = ' "'" ')|' "'",
+                       'CcStateSub6',':S(CcStateSub6)' )
+               SnobolStmt('CcStateSub = template',,':(RETURN)' )
+               )        :(RETURN)
CcStateSetSailEnd

***********************************************************************
*                                                                     *
*               CcPrintSail Edit                                      *
*                                                                     *
***********************************************************************


        DEFINE("CcPrintSail()b,t")                      :(CcPrintSailEnd)
CcPrintSail     CcSep = "If "
        b = SnobolStmt('CcPrint CcPrint = ' QUOTE('('))
CcPrintSailLoop EditInput CcPattern             :F(CcPrintSailDone)
        CcParm "*" REM =
        t = CcSep 'CcCode=' CcValue ' Then ' QUOTE(CcCode)
        t = t IDENT(CcParm,"Integer") '&"="&Cvs(CcParmI)'
        t = t IDENT(CcParm,"String") '&"="&CcParmS'
        b = b SnobolStmt('CcPrint = CcPrint ' CRLF.replace(QUOTE(t)))
        CcSep = ' Else If '             :(CcPrintSailLoop)
CcPrintSailDone b = b SnobolStmt('CcPrint = CcPrint '
+                CRLF.replace(QUOTE('Else "Invalid Control Code"')')),,":(RETURN)")
        EditOutput = SnobolFcn("CcPrint",,b)            :(RETURN)
CcPrintSailEnd
```

III.2 File: Sail Version of CcGenerate Resource

```
        DEFINE("CcVS(A)")       :(CcVSEnd)
CcVS    CcVS = "1" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                       :(RETURN)
CcVSEnd

        DEFINE("CcLM(A)")       :(CcLMEnd)
CcLM    CcLM = "2" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                       :(RETURN)
CcLMEnd
```

```
        DEFINE("CcTM(A)")           :(CcTMEnd)
CcTM    CcTM = "3" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcTMEnd

        DEFINE("CcBM(A)")           :(CcBMEnd)
CcBM    CcBM = "4" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcBMEnd

        DEFINE("CcLIN(A)")          :(CcLINEnd)
CcLIN   CcLIN = "5" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcLINEnd

        DEFINE("CcLA(A)")           :(CcLAEnd)
CcLA    CcLA = "6" "&" A
+                                   :(RETURN)
CcLAEnd

        DEFINE("CcLB(A)")           :(CcLBEnd)
CcLB    CcLB = "7" "&" A
+                                   :(RETURN)
CcLBEnd

        DEFINE("CcUA()")            :(CcUAEnd)
CcUA    CcUA = "8"
+                                   :(RETURN)
CcUAEnd

        DEFINE("CcUB()")            :(CcUBEnd)
CcUB    CcUB = "9"
+                                   :(RETURN)
CcUBEnd

        DEFINE("CcJW(A)")           :(CcJWEnd)
CcJW    CcJW = "10" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcJWEnd

        DEFINE("CcPAD(A)")          :(CcPADEnd)
CcPAD   CcPAD = "11" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcPADEnd

        DEFINE("CcSP(A)")           :(CcSPEnd)
CcSP    CcSP = "12" "&" A
+                                   :(RETURN)
CcSPEnd

        DEFINE("CcEOL()")           :(CcEOLEnd)
CcEOL   CcEOL = "13"
+                                   :(RETURN)
CcEOLEnd

        DEFINE("CcEOP()")           :(CcEOPEnd)
CcEOP   CcEOP = "14"
+                                   :(RETURN)
CcEOPEnd

        DEFINE("CcTAB(A)")          :(CcTABEnd)
CcTAB   CcTAB = "15" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                                   :(RETURN)
CcTABEnd
```

```
        DEFINE("CcQU(A)")       :(CcQUEnd)
CcQU    CcQU = "16" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcQUEnd

        DEFINE("CcOVR(A)")      :(CcOVREnd)
CcOVR   CcOVR = "17" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcOVREnd

        DEFINE("CcSUP(A)")      :(CcSUPEnd)
CcSUP   CcSUP = "18" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcSUPEnd

        DEFINE("CcSUB(A)")      :(CcSUBEnd)
CcSUB   CcSUB = "19" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcSUBEnd

        DEFINE("CcDCP(A)")      :(CcDCPEnd)
CcDCP   CcDCP = "20" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcDCPEnd

        DEFINE("CcUND(A)")      :(CcUNDEnd)
CcUND   CcUND = "21" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcUNDEnd

        DEFINE("CcSL(A)")       :(CcSLEnd)
CcSL    CcSL = "22" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                       :(RETURN)
CcSLEnd

        DEFINE("CcBAK(A)")      :(CcBAKEnd)
CcBAK   CcBAK = "23" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                       :(RETURN)
CcBAKEnd

        DEFINE("CcHD(A)")       :(CcHDEnd)
CcHD    CcHD = "24" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcHDEnd

        DEFINE("CcHN(A)")       :(CcHNEnd)
CcHN    CcHN = "25" "&((" A " LAND '177400) LSH -7)&(" A " LAND '377)"
+                       :(RETURN)
CcHNEnd

        DEFINE("CcBR()")        :(CcBREnd)
CcBR    CcBR = "26"
+                       :(RETURN)
CcBREnd

        DEFINE("CcEOF()")       :(CcEOFEnd)
CcEOF   CcEOF = "27"
+                       :(RETURN)
CcEOFEnd

        DEFINE("CcCMD(A)")      :(CcCMDEnd)
CcCMD   CcCMD = "28" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcCMDEnd
```

```
        DEFINE("CcGR(A)")       :(CcGREnd)
CcGR    CcGR = "29" "&((Length(" A ") LAND '177400) LSH -7)&(Length(" A ") LAND '377)& " A
+                       :(RETURN)
CcGREnd
```

III.3 File: Sail Version of CcDecode Resource

```
        DEFINE("CcNext(ctls,succeed,terminate,fail)")   :(CcNextEnd)
CcNext  CcNext = " Begin Boolean CcErr;Integer CcCode, CcParmI;"
+               "String CcParmS, Sdum;CcErr←False; Sdum←" ctls ";"
+               "If Length(Sdum)=0 Then " terminate
+               " Else Begin CcCode←Lop(Sdum);"
+               "Case CcCode of Begin "
        CcNext = CcNext " [1] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [2] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [3] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [4] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [5] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [6] " " If Length(Sdum)>0 Then CcParmI←Lop(Sdum) Else CcErr←True"
        CcNext = CcNext "; [7] " " If Length(Sdum)>0 Then CcParmI←Lop(Sdum) Else CcErr←True"
        CcNext = CcNext "; [8] "
        CcNext = CcNext "; [9] "
        CcNext = CcNext "; [10] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [11] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [12] " " If Length(Sdum)>0 Then CcParmI←Lop(Sdum) Else CcErr←True"
        CcNext = CcNext "; [13] "
        CcNext = CcNext "; [14] "
        CcNext = CcNext "; [15] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext "; [16] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [17] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [18] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [19] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [20] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [21] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext "; [22] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
```

```
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext ";[23] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext ";[24] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext ";[25] " " If Length(Sdum)>1 Then CcParmI←(Lop(Sdum) LSH 7) "
+               "LOR Lop(Sdum) Else CcErr←True "
        CcNext = CcNext ";[26] "
        CcNext = CcNext ";[27] "
        CcNext = CcNext ";[28] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext ";[29] " " If Length(Sdum)≤1 Then CcErr←True Else Begin "
+               "CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
+               If Length(Sdum)<CcParmI Then CcErr←True Else "
+               "Begin CcParmS←Sdum[1 TO CcParmI];Sdum←Sdum[CcParmI+1 TO ∞] End End "
        CcNext = CcNext " End;"
        CcNext = CcNext "If NOT CcErr Then " succeed " Else " fail " End End "  :(RETURN)
CcNextEnd

        DEFINE("CcPartition(actions)CcSep,p,i") :(CcPartitionEnd)
CcPartition     CcTable = TABLE(30)
        DATA("CcData(value,parm)")
        CcActionPattern = BREAK(":") . CcCode ":" BAL . CcAction ("," | RPOS(0))
        CcTable<"VS"> = CcData(1,"CcParmI")
        CcTable<"LM"> = CcData(2,"CcParmI")
        CcTable<"TM"> = CcData(3,"CcParmI")
        CcTable<"BM"> = CcData(4,"CcParmI")
        CcTable<"LIN"> = CcData(5,"CcParmI")
        CcTable<"LR"> = CcData(6,"CcParmI")
        CcTable<"LB"> = CcData(7,"CcParmI")
        CcTable<"UR"> = CcData(8,"")
        CcTable<"UB"> = CcData(9,"")
        CcTable<"JW"> = CcData(10,"CcParmI")
        CcTable<"PAD"> = CcData(11,"CcParmI")
        CcTable<"SP"> = CcData(12,"CcParmI")
        CcTable<"EOL"> = CcData(13,"")
        CcTable<"EOP"> = CcData(14,"")
        CcTable<"TAB"> = CcData(15,"CcParmI")
        CcTable<"QU"> = CcData(16,"CcParmS")
        CcTable<"OVR"> = CcData(17,"CcParmS")
        CcTable<"SUP"> = CcData(18,"CcParmS")
        CcTable<"SUB"> = CcData(19,"CcParmS")
        CcTable<"DCP"> = CcData(20,"CcParmS")
        CcTable<"UND"> = CcData(21,"CcParmS")
        CcTable<"SL"> = CcData(22,"CcParmI")
        CcTable<"BAK"> = CcData(23,"CcParmI")
        CcTable<"HD"> = CcData(24,"CcParmS")
        CcTable<"HN"> = CcData(25,"CcParmI")
        CcTable<"BR"> = CcData(26,"")
        CcTable<"EOF"> = CcData(27,"")
        CcTable<"CMD"> = CcData(28,"CcParmS")
        CcTable<"GR"> = CcData(29,"CcParmS")
        CcSep =
        CcPartition = " Case CcCode of Begin "
Loop1   actions CcActionPattern =       :F(Done)
        CcCode = TRIMX(CcCode)
        CcDefault = IDENT(CcCode,"*") CcAction  :S(Loop1)
        CcParm = parm(CcTable<CcCode>)
        p =
        p = DIFFER(CcParm) '("' CcParm '")'
        CcAction "cg>" = "|Cc" CcCode p "|"
```

```
Loop2   CcAction "cp>" = CcParm :S(Loop2)
        CcPartition = CcPartition CcSep " [" value(CcTable<CcCode>) "] " CcAction
        CcSep = ";"
        CcTable<CcCode> =       :(Loop1)
Done    DIFFER(CcDefault)       :F(Exit)
        CcArray = CONVERT(CcTable,"ARRAY")      :F(Exit)
        I = 1
Loop3   CcCode = CcArray<I,1>   :F(Exit)
        CcParm = parm(CcTable<CcCode>)
        CcAction = CcDefault
        p =
        p = DIFFER(CcParm) '("' CcParm '")'
        CcAction "cg>" = "!Cc" CcCode p "|"
Loop4   CcAction "cp>" = CcParm :S(Loop4)
        CcPartition = CcPartition CcSep " [" value(CcArray<I,2>) "] " CcAction
        CcSep = ";"
        I = I + 1       :(Loop3)
Exit    CcPartition = CcPartition " End"        :(RETURN)
CcPartitionEnd
```

III.4 File: Sail Version of CcStateSet Resource

```
        DEFINE("CcStateSet(template,separator)")        :(CcStateSetEnd)
CcStateSet      CcStateSet = CcStateSet CcStateSub(template,"VS",1,"Integer*2","7")
        CcStateSet = CcStateSet separator CcStateSub(template,"LM",2,"Integer*2","200")
        CcStateSet = CcStateSet separator CcStateSub(template,"TM",3,"Integer*2","200")
        CcStateSet = CcStateSet separator CcStateSub(template,"BM",4,"Integer*2","200")
        CcStateSet = CcStateSet separator CcStateSub(template,"LIN",5,"Integer*2","55")
        CcStateSet = CcStateSet separator CcStateSub(template,"LA",6,"Integer*1","4")
        CcStateSet = CcStateSet separator CcStateSub(template,"LB",7,"Integer*1","8")
        CcStateSet = CcStateSet separator CcStateSub(template,"JW",10,"Integer*2","8")
        CcStateSet = CcStateSet separator CcStateSub(template,"PAD",11,"Integer*2","8")
        CcStateSet = CcStateSet separator CcStateSub(template,"SL",22,"Integer*2","2200")
        CcStateSet = CcStateSet separator CcStateSub(template,"HD",24,"String","Null")
        CcStateSet = CcStateSet separator CcStateSub(template,"HN",25,"Integer*2","1")
                :(RETURN)
CcStateSetEnd

        DEFINE("CcStateSub(template,code,value,parm,init)")     :(CcStateSubEnd)
CcStateSub      parm "*" REM =
CcStateSub1     template "cc>" = code   :S(CcStateSub1)
CcStateSub2     template "cv>" = value  :S(CcStateSub2)
CcStateSub3     template "ct>" = parm   :S(CcStateSub3)
CcStateSub4     template "ci>" = init   :S(CcStateSub4)
CcStateSub5     template "cg<>" = "!Cc" code '("'       :S(CcStateSub5)
CcStateSub6     template "cg>>" = '")|' :S(CcStateSub6)
        CcStateSub = template   :(RETURN)
CcStateSubEnd
```

III.5 File: Sail Version of CcPrint Resource

```
        DEFINE("CcPrint()")     :(CcPrintEnd)
CcPrint CcPrint CcPrint = "("
        CcPrint = CcPrint "If CcCode=1 Then " '"' "VS" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=2 Then " '"' "LM" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=3 Then " '"' "TM" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=4 Then " '"' "BM" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=5 Then " '"' "LIN" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
```

```
        CcPrint = CcPrint " Else If CcCode=6 Then " '"' "LA" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=7 Then " '"' "LB" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=8 Then " '"' "UA" '"' ""
        CcPrint = CcPrint " Else If CcCode=9 Then " '"' "UB" '"' ""
        CcPrint = CcPrint " Else If CcCode=10 Then " '"' "JW" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=11 Then " '"' "PRD" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=12 Then " '"' "SP" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=13 Then " '"' "EOL" '"' ""
        CcPrint = CcPrint " Else If CcCode=14 Then " '"' "EOP" '"' ""
        CcPrint = CcPrint " Else If CcCode=15 Then " '"' "TAB" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=16 Then " '"' "QU" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=17 Then " '"' "OVR" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=18 Then " '"' "SUP" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=19 Then " '"' "SUB" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=20 Then " '"' "DCP" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=21 Then " '"' "UND" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=22 Then " '"' "SL" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=23 Then " '"' "BAK" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=24 Then " '"' "HD" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=25 Then " '"' "HN" '"' "&" '"' "=" '"' "&Cvs(CcParmI)"
        CcPrint = CcPrint " Else If CcCode=26 Then " '"' "BR" '"' ""
        CcPrint = CcPrint " Else If CcCode=27 Then " '"' "EOF" '"' ""
        CcPrint = CcPrint " Else If CcCode=28 Then " '"' "CMD" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint " Else If CcCode=29 Then " '"' "GR" '"' "&" '"' "=" '"' "&CcParmS"
        CcPrint = CcPrint "Else " '"' "Invalid Control Code" '"' ")"    :(RETURN)
CcPrintEnd
```

IV. Select Program Files

IV.1 File: Select Program Source

```
Begin "TOFF SELECT PROGRAM"
! ************************************************************************;
! *                                                                      *;
! *              Select Pages Program                                    *;
! *                                                                      *;
! *      The source of the Select pages program with the                 *;
! *      I/O and user interface routines deleted.                        *;
! *      Phrases bracketed with braces are macros expressed              *;
! *      as Snobol4 function calls that return string values.            *;
! *                                                                      *;
! ************************************************************************;

Define Acquire="Comment";       Comment Acquire not implemented in Sail;
Acquire "SailExtensions";       Comment Sail Language Extensions;
Acquire "FileUtilities";        Comment Defines Open/Close/Read/Write etc.;
Acquire "StringUtilities";      Comment Defines Index, etc.;


! ************************************************************************;
! *                                                                      *;
! *      The program text for the following functions has                 *;
! *      been deleted.  The functions PagesToKeep and                    *;
! *      PagesToSkip return integers.  If this program is                *;
! *      run interactively, these functions would prompt the             *;
! *      terminal user, parse the response, and check for                *;
! *      validity.  A null response returns a large integer.             *;
! *                                                                      *;
! *      The Input... and Output... functions do the necessary           *;
! *      operating system calls to process input and output              *;
! *      files.                                                          *;
! *                                                                      *;
! ************************************************************************;

Comment Define Operations to Communicate with Terminal User;


Define PagesToSkip="n";
Define PagesToKeep="n";

Comment Define Operations to Read and Write Files;

Define InputInitialize="";
Define InputGetMore="Null";
Define InputTerminate="";
Define OutputInitialize="";
Procedure OutputAppend(integer byte);;
Define OutputTerminate="";


! ************************************************************************;
! *                                                                      *;
! *      Global Variables for the Select Program                         *;
! *                                                                      *;
! *      Buffer:  a string that contains the portion of                  *;
! *               the text-oriented file currently being                 *;
! *               processed.                                             *;
! *                                                                      *;
```

```
! *     Error:  An abort flag                                          *;
! *     Finished:  End of text-oriented file flag                      *;
! *                                                                    *;
! *     SaveICmds: a String of commands encountered on                 *;
! *             skipped pages that must be emitted before              *;
! *             any kept pages are processed.                          *;
! *     Savel...: a set of variables that record the values            *;
! *             of state set control codes while skipping              *;
! *             pages.                                                 *;
! *     Holdl...: a set of variables that preserve the values          *;
! *             of state set control codes while skipping              *;
! *             pages.  If the Savel value of a control code           *;
! *             differs from the Holdl version, the new value          *;
! *             must be emitted before kept pages are processed.       *;
! *                                                                    *;
! ********************************************************************;


String Buffer;
Boolean Error,Finished;

String SaveICmds;
ICcStateSet("ctl Savelccl",";")!;
ICcStateSet("ctl Holdlccl",";")!;


! ********************************************************************;
! *                                                                  *;
! *     Skip N pages, recording relevant commands and changes        *;
! *     in the state set control codes.                              *;
! *                                                                  *;
! ********************************************************************;

Boolean Procedure SKIP(integer N);
    Begin "SKIP"
    Boolean EarlyEof; Integer Pages;
    Pages←0; EarlyEof←False;

    While (Pages<N AND NOT EarlyEof) Do
        !ToffNext(
            "Buffer",
            "!ToffPartition(
                "text:! ignore text;",
                "ctls:!CcNext(
                        "ToffUnit",
                        "!CcPartition(
                            "*: ,
                            !CcStateSet("ccl:Savelccl←cpl",",")!,
                            EOF:EarlyEof←True,
                            CMD:SaveICmds←SaveICmds&!CcCMD("cpl")!,
                            EOP:Pages←Pages+1")!",
                        " ! no action on terminate;",
                        "Error←EarlyEof←Finished←True")!")!",
            "Buffer←Buffer&InputGetMore")!;
    Return(EarlyEof)
    End "SKIP";

! ********************************************************************;
! *                                                                  *;
! *     Keep N pages, transmitting all control codes.                *;
! *                                                                  *;
! ********************************************************************;

Boolean Procedure KEEP(integer N);
    Begin "KEEP"
```

```
    Boolean EarlyEof; Integer Pages;
    EarlyEof←False; Pages←0;
    While (Pages<N AND NOT EarlyEof) Do
        !ToffNext(
            "Buffer",
            "!ToffPartition(
                "text:!ToffText("ToffUnit")!",
                "ctls:Begin String Ctls;
                    !CcNext(
                        "ToffUnit",
                        "!CcPartition(
                            "!CcStateSet("cc⊃:Begin Savelcc⊃←cp⊃;Ctls←Ctls&cg⊃ End",",")!",
                            "s:Ctls←Ctls&cg⊃,
                            EOF:EarlyEof←True")!",
                        "!ToffCtls("Ctls")!",
                        "Error←EarlyEof←Finished←True")!
                        End")!",
            "Buffer←Buffer&InputGetMore")!;
    Return(EarlyEof)
    End "KEEP";

! ***************************************************************;
! *                                                             *;
! *     Alternate between Skipping and Keeping pages from       *;
! *     the input file.  Each time pages are kept, emit         *;
! *     the driver commands that were encountered in the        *;
! *     skipped pages and set any state set control codes       *;
! *     that changed during the skipped pages.                  *;
! *                                                             *;
! ***************************************************************;

Internal Boolean Procedure Select(String OutFile,Infile);
    Begin "SELECT"
    Boolean Once;
    Finished←Once←Error←False;
    Buffer←InputGetMore;
    While NOT Finished Do
        Begin "SKIP/KEEP LOOP"
        Integer n;

        n←PagesToSkip;
        If n=0
           Then Begin Finished←Once; Once←True End
           Else Begin "SKIP SECTION OF LOOP"
                !CcStateSet("Savelcc⊃←Holdlcc⊃",";")!;
                Finished←SKIP(n)
                End "SKIP SECTION OF LOOP";
        If Finished Then Done;

        n←PagesToKeep;
        If n=0
           Then Begin Finished←Once; Once←True End
           Else Begin "KEEP SECTION OF LOOP"
                String Ctls;
                !CcStateSet("If NOT Savelcc⊃←Holdlcc⊃ Then Ctls←Ctls&cg<⊃Savelcc⊃cg>⊃",";")!;
                !ToffCtls("Savelcmds&Ctls")!;
                Finished←KEEP(n)
                End "KEEP SECTION OF LOOP"
        End "SKIP/KEEP LOOP";

    !ToffCtls("!CcEOF!")!;
    Return(NOT Error)
    End "SELECT"
    !
End "TOFF SELECT PROGRAM"
```

IV.2 File: Select Program Sail Text

```
Begin "TOFF SELECT PROGRAM"
! ************************************************************************;
! *                                                                      *;
! *              Select Pages Program                                    *;
! *                                                                      *;
! *      The source of the Select pages program with the                 *;
! *      I/O and user interface routines deleted.                        *;
! *      Phrases bracketed with braces are macros expressed              *;
! *      as Snobol4 function calls that return string values.            *;
! *                                                                      *;
! ************************************************************************;
Define Acquire="Comment";
Comment Acquire not implemented in Sail;
Acquire "SailExtensions";
Comment Sail Language Extensions;
Acquire "FileUtilities";
Comment Defines Open/Close/Read/Write etc.;
Acquire "StringUtilities";
Comment Defines Index, etc.;
! ************************************************************************;
! *                                                                      *;
! *      The program text for the following functions has                 *;
! *      been deleted.  The functions PagesToKeep and                    *;
! *      PagesToSkip return integers.  If this program is                *;
! *      run interactively, these functions would prompt the             *;
! *      terminal user, parse the response, and check for                *;
! *      validity.  A null response returns a large integer.             *;
! *                                                                      *;
! *      The Input... and Output... functions do the necessary           *;
! *      operating system calls to process input and output              *;
! *      files.                                                          *;
! *                                                                      *;
! ************************************************************************;
Comment Define Operations to Communicate with Terminal User;
Define PagesToSkip="n";
Define PagesToKeep="n";
Comment Define Operations to Read and Write Files;
Define InputInitialize="";
Define InputGetMore="Null";
Define InputTerminate="";
Define OutputInitialize="";
Procedure OutputAppend(integer byte);
;
Define OutputTerminate="";
! ************************************************************************;
! *                                                                      *;
! *      Global Variables for the Select Program                         *;
! *                                                                      *;
! *      Buffer:  a string that contains the portion of                  *;
! *               the text-oriented file currently being                 *;
! *               processed.                                             *;
! *                                                                      *;
! *      Error:  An abort flag                                           *;
! *      Finished:  End of text-oriented file flag                       *;
! *                                                                      *;
! *      SaveICmds: a String of commands encountered on                  *;
! *               skipped pages that must be emitted before              *;
! *               any kept pages are processed.                          *;
! *      SaveI...: a set of variables that record the values             *;
! *               of state set control codes while skipping              *;
```

```
! *             pages.                                                *;
! *     Hold!...: a set of variables that preserve the values         *;
! *             of state set control codes while skipping             *;
! *             pages.  If the Save! value of a control code          *;
! *             differs from the Hold! version, the new value         *;
! *             must be emitted before kept pages are processed.      *;
! *                                                                   *;
! ********************************************************************;
String Buffer;
Boolean Error,Finished;
String Save!Cmds;
Integer Save!VS;
Integer Save!LM;
Integer Save!TM;
Integer Save!BM;
Integer Save!LIN;
Integer Save!LA;
Integer Save!LB;
Integer Save!JW;
Integer Save!PAD;
Integer Save!SL;
String Save!HD;
Integer Save!HN;
Integer Hold!VS;
Integer Hold!LM;
Integer Hold!TM;
Integer Hold!BM;
Integer Hold!LIN;
Integer Hold!LA;
Integer Hold!LB;
Integer Hold!JW;
Integer Hold!PAD;
Integer Hold!SL;
String Hold!HD;
Integer Hold!HN;
! ********************************************************************;
! *                                                                   *;
! *     Skip N pages, recording relevant commands and changes         *;
! *     in the state set control codes.                               *;
! *                                                                   *;
! ********************************************************************;
Boolean Procedure SKIP(integer N);
  Begin "SKIP" Boolean EarlyEof;
  Integer Pages;
  Pages←0;
  EarlyEof←False;
  While (Pages<N AND NOT EarlyEof) Do
    Begin Define ToffText=False,ToffCtls=True;
    String ToffUnit;
    Boolean ToffGotIt,ToffType;
    Integer ToffPtr,ToffCnt;
    External Integer Procedure Index(string x,y);
    ToffGotIt←False;
    If Length(Buffer)>0
      Then If Buffer[1 For 1]=0
        Then
          Begin If Length(Buffer)>1
            Then
              Begin ToffUnit←Lop(Buffer);
              ToffCnt←(Lop(Buffer) LSH 7) LOR Lop(Buffer);
              If Length(Buffer)≥ToffCnt
                Then
                  Begin ToffUnit←Buffer[1 To ToffCnt];
                  Buffer←Buffer[ToffCnt To ∞];
                  ToffType←ToffCtls;
```

```
                ToffGotIt←True
                End
            End
        End
    Else If (ToffPtr←Index(Buffer,8))>8
        Then
          Begin ToffUnit←Buffer[1 For ToffPtr-1];
          Buffer←Buffer[ToffPtr to ∞];
          ToffType←ToffText;
          ToffGotIt←True
          End;
If ToffGotIt
  Then If ToffType=ToffText
    Then ! ignore text;
    Else
      Begin Boolean CcErr;
      Integer CcCode, CcParmI;
      String CcParmS, Sdum;
      CcErr←False;
      Sdum←ToffUnit;
      If Length(Sdum)=8
        Then  ! no action on terminate;
        Else
          Begin CcCode←Lop(Sdum);
          Case CcCode of
            Begin  [1]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [2]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [3]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [4]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [5]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [6]  If Length(Sdum)>8
              Then CcParmI←Lop(Sdum)
              Else CcErr←True;
            [7]  If Length(Sdum)>8
              Then CcParmI←Lop(Sdum)
              Else CcErr←True;
            [8] ;
            [9] ;
            [10]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [11]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [12]  If Length(Sdum)>8
              Then CcParmI←Lop(Sdum)
              Else CcErr←True;
            [13] ;
            [14] ;
            [15]  If Length(Sdum)>1
              Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
              Else CcErr←True ;
            [16]  If Length(Sdum)≤1
              Then CcErr←True
              Else
```

```
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[17]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[18]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[19]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[20]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[21]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[22]  If Length(Sdum)>1
  Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
```

```
    Else CcErr←True ;
  [23]  If Length(Sdum)>1
    Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
    Else CcErr←True ;
  [24]  If Length(Sdum)≤1
    Then CcErr←True
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End ;
  [25]  If Length(Sdum)>1
    Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
    Else CcErr←True ;
  [26] ;
  [27] ;
  [28]  If Length(Sdum)≤1
    Then CcErr←True
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End ;
  [29]  If Length(Sdum)≤1
    Then CcErr←True
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End
  End;
If NOT CcErr
  Then  Case CcCode of
    Begin  [1] Save!VS←CcParmI;
    [2] Save!LM←CcParmI;
    [3] Save!TM←CcParmI;
    [4] Save!BM←CcParmI;
    [5] Save!LIN←CcParmI;
    [6] Save!LA←CcParmI;
    [7] Save!LB←CcParmI;
    [10] Save!JW←CcParmI;
    [11] Save!PAD←CcParmI;
    [22] Save!SL←CcParmI;
    [24] Save!HD←CcParmS;
    [25] Save!HN←CcParmI;
    [27] EarlyEof←True;
    [28] Save!Cmds←Save!Cmds&28&((Length(CcParmS) LAND '177400) LSH -7)&
                  (Length(CcParmS) LAND '377)& CcParmS;
    [14] Pages←Pages+1;
    [17]  ;
    [26]  ;
    [21]  ;
    [18]  ;
```

```
                    [16]  ;
                    [20]  ;
                    [23]  ;
                    [29]  ;
                    [15]  ;
                    [13]  ;
                    [12]  ;
                    [8]   ;
                    [9]   ;
                    [19]
                    End
                  Else Error←EarlyEof←Finished←True
                End
            End
        Else Buffer←Buffer&InputGetMore
      End ;
    Return(EarlyEof)
    End "SKIP";
! ********************************************************************;
! *                                                                  *;
! *     Keep N pages, transmitting all control codes.                *;
! *                                                                  *;
! ********************************************************************;
Boolean Procedure KEEP(Integer N);
  Begin "KEEP" Boolean EarlyEof;
  Integer Pages;
  EarlyEof←False;
  Pages←0;
  While (Pages<N AND NOT EarlyEof) Do
    Begin Define ToffText=False,ToffCtls=True;
    String ToffUnit;
    Boolean ToffGotIt,ToffType;
    Integer ToffPtr,ToffCnt;
    External Integer Procedure Index(string x,y);
    ToffGotIt←False;
    If Length(Buffer)>0
      Then If Buffer[1 For 1]=0
        Then
          Begin If Length(Buffer)>1
            Then
              Begin ToffUnit←Lop(Buffer);
              ToffCnt←(Lop(Buffer) LSH 7) LOR Lop(Buffer);
              If Length(Buffer)≥ToffCnt
                Then
                  Begin ToffUnit←Buffer[1 To ToffCnt];
                  Buffer←Buffer[ToffCnt To ∞];
                  ToffType←ToffCtls;
                  ToffGotIt←True
                  End
              End
          End
        Else If (ToffPtr←Index(Buffer,0))>0
          Then
            Begin ToffUnit←Buffer[1 For ToffPtr-1];
            Buffer←Buffer[ToffPtr to ∞];
            ToffType←ToffText;
            ToffGotIt←True
            End;
    If ToffGotIt
      Then If ToffType=ToffText
        Then
          Begin String Sdum;
          Sdum←ToffUnit;
          While Length(Sdum)>0 Do OutputAppend(Lop(Sdum))
          End
```

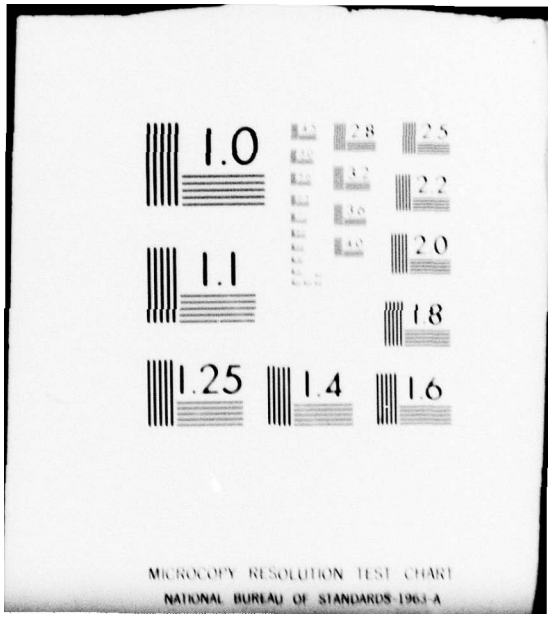
```
Else
  Begin String Ctls;
    Begin Boolean CcErr;
    Integer CcCode, CcParmI;
    String CcParmS, Sdum;
    CcErr←False;
    Sdum←ToffUnit;
    If Length(Sdum)=0
      Then
        Begin String Sdum;
        Sdum ← Ctls;
        OutputAppend(0);
        OutputAppend((Length(Sdum) AND '177400) LSH -7);
        OutputAppend(Length(Sdum) AND '177);
        While Length(Sdum)>0 Do OutputAppend(Lop(Sdum))
        End
      Else
        Begin CcCode←Lop(Sdum);
        Case CcCode of
          Begin  [1]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [2]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [3]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [4]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [5]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [6]  If Length(Sdum)>0
            Then CcParmI←Lop(Sdum)
            Else CcErr←True;
          [7]  If Length(Sdum)>0
            Then CcParmI←Lop(Sdum)
            Else CcErr←True;
          [8] ;
          [9] ;
          [10]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [11]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [12]  If Length(Sdum)>0
            Then CcParmI←Lop(Sdum)
            Else CcErr←True;
          [13] ;
          [14] ;
          [15]  If Length(Sdum)>1
            Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
            Else CcErr←True ;
          [16]  If Length(Sdum)≤1
            Then CcErr←True
            Else
              Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
              If Length(Sdum)<CcParmI
                Then CcErr←True
                Else
                  Begin CcParmS←Sdum[1 TO CcParmI];
                  Sdum←Sdum[CcParmI+1 TO ∞]
```

```
        End
    End ;
[17]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[18]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[19]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[20]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[21]  If Length(Sdum)≤1
  Then CcErr←True
  Else
    Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
    If Length(Sdum)<CcParmI
      Then CcErr←True
      Else
        Begin CcParmS←Sdum[1 TO CcParmI];
        Sdum←Sdum[CcParmI+1 TO ∞]
        End
    End ;
[22]  If Length(Sdum)>1
  Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
  Else CcErr←True ;
[23]  If Length(Sdum)>1
  Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
  Else CcErr←True ;
[24]  If Length(Sdum)≤1
  Then CcErr←True
```

AD-A070 955 CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER --ETC F/G 9/2
THE REPRESENTATION OF FAMILIES OF SOFTWARE SYSTEMS.(U)
APR 79 L W COOPRIDER F44620-73-C-0074
UNCLASSIFIED CMU-CS-79-116 AFOSR-TR-79-0732 NL

3 OF 3
AD A070955

END
DATE FILMED
8-79
DDC


1.0
1.1
1.25
1.4
1.6
2.8
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End ;
  [25]  If Length(Sdum)>1
    Then CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum)
    Else CcErr←True ;
  [26] ;
  [27] ;
  [28]  If Length(Sdum)≤1
    Then CcErr←True
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End ;
  [29]  If Length(Sdum)≤1
    Then CcErr←True
    Else
      Begin CcParmI←(Lop(Sdum) LSH 7) LOR Lop(Sdum);
      If Length(Sdum)<CcParmI
        Then CcErr←True
        Else
          Begin CcParmS←Sdum[1 TO CcParmI];
          Sdum←Sdum[CcParmI+1 TO ∞]
          End
      End
  End;
If NOT CcErr
  Then  Case CcCode of
    Begin  [1]
      Begin Save!VS←CcParmI;
      Ctls←Ctls&I&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
      End;
    [2]
      Begin Save!LM←CcParmI;
      Ctls←Ctls&2&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
      End;
    [3]
      Begin Save!TM←CcParmI;
      Ctls←Ctls&3&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
      End;
    [4]
      Begin Save!BM←CcParmI;
      Ctls←Ctls&4&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
      End;
    [5]
      Begin Save!LIN←CcParmI;
      Ctls←Ctls&5&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
      End;
    [6]
      Begin Save!LA←CcParmI;
      Ctls←Ctls&6&CcParmI
      End;
    [7]
      Begin Save!LB←CcParmI;
```

```
                    Ctls←Ctls&74&CcParmI
                    End;
                  [10]
                    Begin SaveIJW←CcParmI;
                    Ctls←Ctls&108&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
                    End;
                  [11]
                    Begin SaveIPRD←CcParmI;
                    Ctls←Ctls&118&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
                    End;
                  [22]
                    Begin SaveISL←CcParmI;
                    Ctls←Ctls&228&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
                    End;
                  [24]
                    Begin SaveIND←CcParmS;
                    Ctls←Ctls&248&((Length(CcParmS) LAND '177400) LSH -7)&
                                          (Length(CcParmS) LAND '377)& CcParmS
                    End;
                  [25]
                    Begin SaveIHN←CcParmI;
                    Ctls←Ctls&258&((CcParmI LAND '177400) LSH -7)&(CcParmI LAND '377)
                    End;
                  [17]  ;
                  [26]  ;
                  [21]  ;
                  [18]  ;
                  [16]  ;
                  [27]  ;
                  [20]  ;
                  [23]  ;
                  [29]  ;
                  [28]  ;
                  [15]  ;
                  [13]  ;
                  [12]  ;
                  [8]   ;
                  [9]   ;
                  [19]  ;
                  [14]
                  End
                Else Error←EarlyEof←Finished←True
              End
            End
          End
      Else Buffer←Buffer&InputGetMore
    End ;
  Return(EarlyEof)
  End "KEEP";
! ********************************************************************;
! *                                                                  *;
! *     Alternate between Skipping and Keeping pages from            *;
! *     the input file.  Each time pages are kept, emit              *;
! *     the driver commands that were encountered in the             *;
! *     skipped pages and set any state set control codes            *;
! *     that changed during the skipped pages.                       *;
! *                                                                  *;
! ********************************************************************;
Internal Boolean Procedure Select(String OutFile,InFile);
  Begin "SELECT" Boolean Once;
  Finished←Once←Error←False;
  Buffer←InputGetMore;
  While NOT Finished Do
    Begin "SKIP/KEEP LOOP" Integer n;
    n←PagesToSkip;
```

```
    If n=0
      Then
        Begin Finished←Once;
        Once←True
        End
      Else
        Begin "SKIP SECTION OF LOOP"Save!VS←Hold!VS;
        Save!LM←Hold!LM;
        Save!TM←Hold!TM;
        Save!BM←Hold!BM;
        Save!LIN←Hold!LIN;
        Save!LA←Hold!LA;
        Save!LB←Hold!LB;
        Save!JW←Hold!JW;
        Save!PAD←Hold!PAD;
        Save!SL←Hold!SL;
        Save!HD←Hold!HD;
        Save!HN←Hold!HN;
        Finished←SKIP(n)
        End "SKIP SECTION OF LOOP";
    If Finished
      Then Done;
    n←PagesToKeep;
    If n=0
      Then
        Begin Finished←Once;
        Once←True
        End
      Else
        Begin "KEEP SECTION OF LOOP" String Ctls;
        If NOT Save!VS=Hold!VS
          Then Ctls←Ctls&1&((Save!VS LAND '177400) LSH -7)&(Save!VS LAND '377);
        If NOT Save!LM=Hold!LM
          Then Ctls←Ctls&2&((Save!LM LAND '177400) LSH -7)&(Save!LM LAND '377);
        If NOT Save!TM=Hold!TM
          Then Ctls←Ctls&3&((Save!TM LAND '177400) LSH -7)&(Save!TM LAND '377);
        If NOT Save!BM=Hold!BM
          Then Ctls←Ctls&4&((Save!BM LAND '177400) LSH -7)&(Save!BM LAND '377);
        If NOT Save!LIN=Hold!LIN
          Then Ctls←Ctls&5&((Save!LIN LAND '177400) LSH -7)&(Save!LIN LAND '377);
        If NOT Save!LA=Hold!LA
          Then Ctls←Ctls&6&Save!LA;
        If NOT Save!LB=Hold!LB
          Then Ctls←Ctls&7&Save!LB;
        If NOT Save!JW=Hold!JW
          Then Ctls←Ctls&10&((Save!JW LAND '177400) LSH -7)&(Save!JW LAND '377);
        If NOT Save!PAD=Hold!PAD
          Then Ctls←Ctls&11&((Save!PAD LAND '177400) LSH -7)&(Save!PAD LAND '377);
        If NOT Save!SL=Hold!SL
          Then Ctls←Ctls&22&((Save!SL LAND '177400) LSH -7)&(Save!SL LAND '377);
        If NOT Save!HD=Hold!HD
          Then Ctls←Ctls&24&((Length(Save!HD) LAND '177400) LSH -7)&
                                (Length(Save!HD) LAND '377)& Save!HD;
        If NOT Save!HN=Hold!HN
          Then Ctls←Ctls&25&((Save!HN LAND '177400) LSH -7)&(Save!HN LAND '377);
          Begin String Sdum;
          Sdum ← Save!Cmds&Ctls;
          OutputAppend(0);
          OutputAppend((Length(Sdum) AND '177400) LSH -7);
          OutputAppend(Length(Sdum) AND '177);
          While Length(Sdum)>0 Do OutputAppend(Lop(Sdum))
          End ;
        Finished←KEEP(n)
        End "KEEP SECTION OF LOOP"
    End "SKIP/KEEP LOOP";
```

```
    Begin String Sdum;
    Sdum ← 27;
    OutputAppend(0);
    OutputAppend((Length(Sdum) AND '177400) LSH -7);
    OutputAppend(Length(Sdum) AND '177);
    While Length(Sdum)>0 Do OutputAppend(Lop(Sdum))
    End ;
  Return(NOT Error)
  End "SELECT";
End "TOFF SELECT PROGRAM"
```

References

[Alme77] Guy Almes and George Robertson. *An Extensible File System for Hydra.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[Ambl77] A.L. Ambler et al. GYPSY: A Language for Specification and Implementation of Verifiable Programs. *Proceedings of the ACM Conference on Language Design for Reliable Software, SIGPLAN Notices 12,3*:1-10, March 1977.

[Balz76] Robert Balzer, David Wile and Neil Goldman. On the Transformational Implementation Approach to Programming. *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[Bela71] L. Belady and M. Lehman. *Programming System Dynamics.* Technical Report RC 3546, IBM Thomas J. Watson Research Center, 1971.

[Bian76] M.H. Bianchi and J.L. Wood. A User's Viewpoint on the Programmer's Workbench. *Proceedings of the 2nd International Conference on Software* Engineering, 1976.

[Boeh73] Barry Boehm. The High Cost of Software. *Proceedings of the Symposium on the High Cost of Software*, 1973.

[Boeh75] Barry Boehm. Some Experience with Automated Aids to the Design of Large Scale Reliable Software. *Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10,6*:105-113, June 1975.

[Brat75] H. Bratman and T. Court. The Software Factory. *Computer* 8(5), May 1975.

[Brav74] Harry Braverman. *Labor and Monopoly Capital: The Degradation of Work in the Twentieth Century.* Monthly Review Press, 1974.

[Broo75] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software* Engineering. Addison-Wesley, 1975.

[Brow70] H.B. Brown. The Clear/Caster System. *Software Engineering Techniques, Nato Conference on Software Engineering*, 1970.

[Carp75] Loren C. Carpenter and Leonard L. Tripp. Software Design Validation Tool. *Proceedings of the International Conference on Reliable Software, SIGPLAN Notices 10,6*, 1975.

[Catt78] R.G.G. Cattell. *Formalization and Automatic Derivation of Code Generators.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1978.

[Dahl72] Ole-Johan Dahl, Edsger W. Dijkstra and C.A.R. Hoare. *Structured* Programming. Academic Press, 1972.

[DeJo73] S.P. DeJong. *The System Building System.* Technical Report RC 4486, Thomas J. Watson Research Center, 1973.

[DeRe76] Frank DeRemer and H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering* 2(2), June 1976.

[Digi73] Digital Equipment Corporation. *Batch-11/Dos-11 Assembler (Macro-11).* Digital Equipment Corporation, 1973.

[Digi72] Digital Equipment Corporation. *DecSystem10 Users Handbook.* Digital Equipment Corporation, 1972.

[Dijk72] Edsger W. Dijkstra. Notes on Structured Programming, in Ole-Johan Dahl, Edsger W. Dijkstra and &C.A.R. Hoare, *Structured Programming* Academic Press, 1972.

[Dijk68] Edsger W. Dijkstra. The Structure of the "THE"-Multiprogramming System. *Communications of the ACM* 11(5):341-346, May 1968.

[DoD77] Department of Defense. Requirements for High Order Computer Programming Languages. *SIGPLAN Notices* 12(12):39-54, December 1977.

[Dolo76a] T.A. Dolotta and J.R. Mashey. An Introduction to the Programmer's Workbench. *Proceedings of the 2nd International Conference on Software* Engineering, 1976.

[Dolo76b] T.A. Dolotta and J.S. Licwinko. The Leap Load and Test Driver. *Proceedings of the 2nd International Conference on Software Engineering,* 1976.

[Dund75] Allan Dundes and Carl R. Pagter. Project Swing. *Urban Folklore from the Paperwork Empire, American Folklore Society Memoir Series* 62(168):975, 1975.

[Erma77] Lee Erman and Victor Lesser. *System Engineering Techniques for Artificial Intelligence Programs.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[Flon75a] Lawrence Flon. On Research in Structured Programming. *SIGPLAN Notices* 10(10), October 1975.

[Flon75b] Lawrence Flon. *Program Design With Abstract Data Types.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1975.

[Flon77] Lawrence Flon. *On the Design and Verification of Operating Systems.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[Gesc77] Charles Geschke et al. Early Experience With Mesa. *Communications of the ACM* 20(8):540-552, August 1977.

[Gris72] Ralph E. Griswold. *The Macro Implementation of Snobol4, A Case Study of Machine-Independent Software Development.* W.H. Freeman, 1972.

[Habe76] A. Nico Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the* ACM 19(5):266-272, May 1976.

[Habe77] A. Nico Habermann. On System Design and Maintenance Control (private communication).

[IBM72] IBM Corporation. *System/360 Operating System System Generation*, GC28-6554-11. IBM Corporation, 1972.

[Irvi77] C.A. Irvine and John W. Brackett. Automated Software Engineering Through Structured Data Management. *IEEE Transactions on Software Engineering* 3(1), January 1977.

[Jens74] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, 1974.

[Kern74] B.W. Kernighan and P.J. Plauger. *The Elements of Programming Style*. McGraw-Hill, 1974.

[Kern76] B.W. Kernighan and P.J. Plauger. *Software Tools*. Addison-Wesley, 1976.

[Knud76] D.B. Knudsen, A. Barofsky and L.R. Satz. A Modification Request Control System. *Proceedings of the 2nd International Conference on Software Engineering*, 1976.

[Krut75] Rudy Krutar. *Flexors*. Technical Report, Computer Science Department, Carnegie-Mellon University, 1975.

[Lamp77] Butler W. Lampson, J.J. Horning, Ralph L. London, J.G. Mitchell and G.J. Popek. Report on the Programming Language Euclid. *SIGPLAN Notices* 12(2), February 1977.

[Levi77] Roy Levin, David Jefferson and Joseph M. Newcomer. *C.mmp Linker Reference Manual*. Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[Lisk76] Barbara H. Liskov. An Introduction to CLU, in S.A. Schuman, editor, *New Directions in Algorithmic Languages - 1975* IRIA, 1976.

[Lisk72] Barbara H. Liskov. The Design of the VENUS Operating System. *Communications of the ACM* 15(3):144-149, March 1972.

[Lisk74a] Barbara H. Liskov. *Implications of the Implementation of Libraries and* Directories. Technical Report CLU Design Note 15, Project MAC, Massachusetts Institute of Technology, 1974.

[Lisk74b] Barbara H. Liskov. *Description Units, Libraries and Directories, CLU Design Note 14*. Technical Report, Project MAC, Massachusetts Institute of Technology, 1974.

[Mao48] Mao ZeDong. *Selected Readings from the Works of Mao ZeDong*. China Foreign Languages Press, Beijing, 1948.

[Mash76a] J.R. Mashey. Using a Command Language as a High-Level Programming Language. *Proceedings of the 2nd International Conference on Software* Engineering, 1976.

[Mash76b] J.R. Mashey and D.W. Smith. Documentation Tools and Techniques. *Proceedings of the 2nd International Conference on Software Engineering*,

1976.

[Nels77] Loose Bruce Nelson. Tingle: or What to do Until the Computer Comes (private communication).

[Neum74] Peter G. Neumann et al. On the Design of a Provably Secure Operating System. *Proceedings of the IRIA Workshop on Protection in Operating Systems, Paris*:161-175, August 1974.

[Newc74] Joseph M. Newcomer. *BH: A General Information Organization Program.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1974.

[Newe77] Allen Newell. (private communication).

[Newe72] Allen Newell and Herbert Simon. *Human Problem Solving.* Prentice-Hall, 1972.

[Orga] Eliot I. Organick. *The Multics System: An Examination of its Structure.* MIT Press, .

[Parn76a] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering* 2(1):1-8, March 1976.

[Parn72a] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM* 15(12):1053-1058, December 1972.

[Parn72b] David L. Parnas. Some Conclusions From an Experiment in Software Engineering Techniques. *Proceedings Fall Joint Computer Conference, Vol* 41:325-329, 1972.

[Parn74] David L. Parnas. On a "Buzzword": Hierarchical Structure. *Proceedings of the IFIPS Congress 74*, 1974.

[Parn77] David L. Parnas. Building Reliable Software Systems in Blowhard. *Software Engineering Notes* 2(3), April 1977.

[Parn72c] David L. Parnas and D.P. Siewiorek. *Use of the Concept of Transparency in the Design of Hierarchically Structured Systems.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1972.

[Parn76b] David L. Parnas. *Some Hypotheses about the "Uses" Hierarchy for Operating Systems.* Technical Report, Technische Hochschule Darmstadt, Fachbereich Informatik, 1976.

[Parn72d] David L. Parnas. A Technique for Software Module Specification With Examples. *Communications of the ACM* 15(5):330-336, May 1972.

[Reid77] Brian K. Reid. A File System for Program Development, Documentation and Maintenance (private communication).

[Rein77] Andrew Reiner and Joseph M. Newcomer. *Hydra User's Manual.* Technical Report, Computer Science Department, Carnegie-Mellon University, 1977.

[Reis76] John F. Reiser. *Sail.* Technical Report AIM-289, Stanford Artificial

Intelligence Laboratory, 1976.

[Rich74] D.M. Richie and K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM* 17(7):365ff, July 1974.

[Roch74] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering* 1(4):364-370, December 1974.

[Ross77] Doug T. Ross. Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering* 3(1):16-33, January 1977.

[Saxe76] A.R. Saxena. *A Verified Specification of a Hierarchical Operating System.* Technical Report, Stanford University, 1976.

[Schw79] Bob Schwanke. *Representation Management in Programming Languages and Operating Systems (Ph.D. Thesis in preparation).* Technical Report, Computer Science Department, Carnegie-Mellon University, 1979.

[Shaw77] Mary Shaw, William A. Wulf and Ralph L. London. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. *Communications of the ACM* 20(8):553-560, August 1977.

[Tayl11] Frederick W. Taylor. *Principles of Scientific Management.* New York, 1911.

[Teic77] D. Teichroew and E.A. Hershey, III. PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Processing Systems. *IEEE Transactions on Software Engineering* 3(1):41-48, January 1977.

[Thom76] Joseph Thomas. *Module Interconnection in Programming Systems Supporting Abstraction.* Technical Report CS-16, Brown University, 1976.

[Tich80] Walter Tichy. *Towards Support for the Construction and Maintenance of Modular Programmed Systems (in progress).* Technical Report, Computer Science Department, Carnegie-Mellon University, 1980.

[Tich77] Walter Tichy. Intercol: A Module Interconnection Language (private communication).

[Wein71] Gerald M. Weinberg. *The Psychology of Computer Programming.* Van Nostrand Reinhold, 1971.

[Weiz76] Joseph Weizenbaum. *Computer Power and Human Reason.* W.H. Freeman, 1976.

[Whit77] John R. White and Richard K. Anderson. Supporting the Structured Development of Complex PL/I Software Systems. *Software - Practice and Experience* 7:279-293, 1977.

[Wirt77] Niklaus Wirth. Modula: A Language for Modular Programming. *Software - Practice and Experience* 7(1):3-35, 1977.

[Wirt71] Niklaus Wirth. The Design of a Pascal Compiler. *Software - Practice and Experience* 1:309-333, 1971.

[Wulf76] William A. Wulf, Ralph L. London and Mary Shaw. An Introduction to the Construction and Verification of Alphard Programs. *IEEE Transactions on Software Engineering* 2(4):253-265, December 1976.

[Wulf70] William A. Wulf et al. *Bliss Reference Manual.* Digital Equipment Corporation, 1970.

[Wulf72] William A. Wulf et al. *Bliss-11 Programmer's Manual.* Digital Equipment Corporation, 1972.

[Zelk78] Marvin V. Zelkowitz. Perspectives on Software Engineering. *Computing* Surveys 10(2), June 1978.